CIS 505: Software Systems

OS Overview -- Processes and Threads

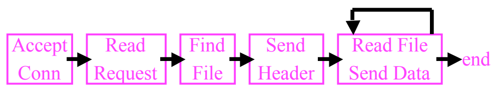
Insup Lee
Department of Computer and Information Science
University of Pennsylvania

CIS 505, Spring 2007

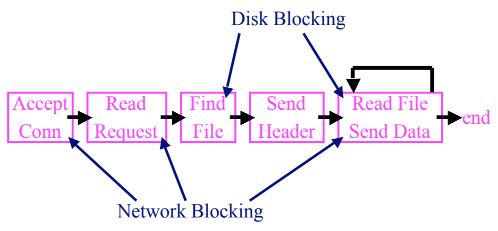
Processes

- What is process?
  - Program in execution
- Why need multiple processes?
  - Overlap between I/O and computation
  - Time sharing
  - Multiple CPU allocations

Control flow of a Web server



Blocking steps



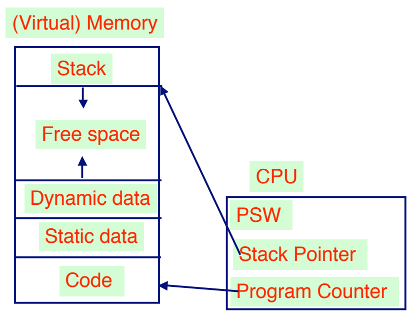
Event-driven programming

- "reactive" environments
  - Windowing systems
  - Network aware programs
  - Device drivers
- What is an event? Some kind of notification
  - Interrupts
  - Signals
  - Polling (via poll/select/etc)
  - Callback (via function pointer)
- Good performance
  - No processes!
- Lots of "if then else"
- So why processes/threads?

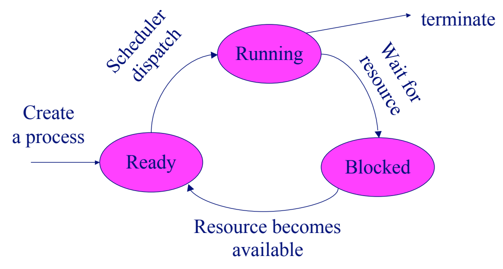
Contents of a process

- A process includes
  - An address space (code, data)
  - A resource container (OS resource, accounting)
  - A "thread of control", which defines where the process is currently executing (basically, PC, registers, and stack)

Process Snapshot



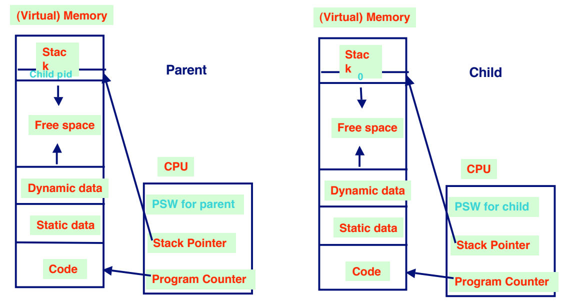
Process State Transition



The Fork System Call

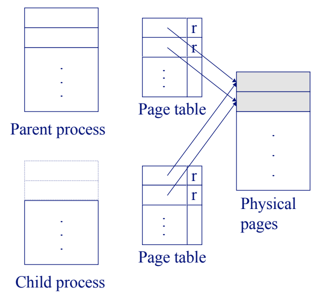
- The **fork()** system call creates a "clone" of the calling process.
- Identical in every respect except
  - the parent process is returned a non-zero value (namely, the process id of the child)
  - the child process is returned zero.
- The process id returned to the parent can be used by parent in a **wait** or **kill** system call.

Snapshots after fork()



Copy-On-Write

- Child's virtual address space uses the same page mapping as parent's
- Make all pages read-only
- Make child process ready
- On a read, nothing happens
- On a write, generates an access fault
  - map to a new page frame
  - copy the page over
  - restart the instruction



Example using fork

```
#include <unistd.h>
main(){
  pid_t pid;
  printf("Just one process so far\n");
  pid = fork();
  if (pid == 0) /* code for child */
     printf("I'm the child\n");
  else if (pid > 0) /* code for parent */
     printf("The parent, child pid =%d\n",
            pid);
  else /* error handling */
     printf("Fork returned error code\n");
}
```

Sample Question

```
main(){
  int x=0;
  fork();
  x++;
  printf("The value of x is %d\n",
x);
}
```

What will be the output?

Spawning Applications

fork() is typically used in conjunction with exec (or variants)

```
pid_t pid;
if ( ( pid = fork() ) == 0 ) {
   /* child code: replace executable image */
   execv( "/usr/games/tetris", "-easy" )
} else {
   /* parent code: wait for child to terminate */
   wait( &status )
}
```

exec System Call

A family of routines, **execl**, **execv**, ..., all eventually make a call to **execve**.

execve(program_name, arg1, arg2, ..., environment)

- text and data segments of current process replaced with those of **program_name**
- stack reinitialized with parameters
- open file table of current process remains intact
- the last argument can pass environment settings
- as in example, **program_name** is actually path name of executable file containing program

Note: unlike subroutine call, there is no return after this call. That is, the program calling exec is gone forever!

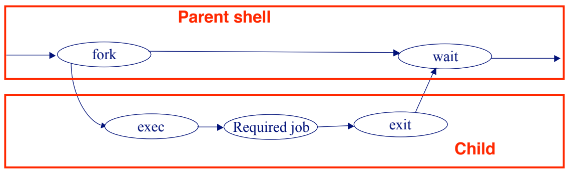
Parent-Child Synchronization

- **exit(status)** - executed by a child process when it wants to terminate. Makes **status** (an integer) available to parent.
- **wait(&status)** - suspends execution of process until *some* child process terminates
  - **status** indicates reason for termination
  - return value is process-id of terminated child
- **waitpid (pid, &status, options)**
  - pid can specify a specific child
  - Options can be to wait or to check and proceed

Process Termination

- Besides being able to terminate itself with **exit**, a process can be killed by another process using **kill**:
  - **kill(pid, sig)** - sends signal **sig** to process with process-id **pid**. One signal is **SIGKILL** (terminate the target process immediately).
- When a process terminates, all the resources it owns are reclaimed by the system:
  - "process control block" reclaimed
  - its memory is deallocated
  - all open files closed and Open File Table reclaimed.
- Note: a process can kill another process only if:
  - it belongs to the same user
  - super user

How shell executes a command



- when you type a command, the shell forks a clone of itself
- the child process makes an exec call, which causes it to stop executing the shell and start executing your command
- the parent process, still running the shell, waits for the child to terminate

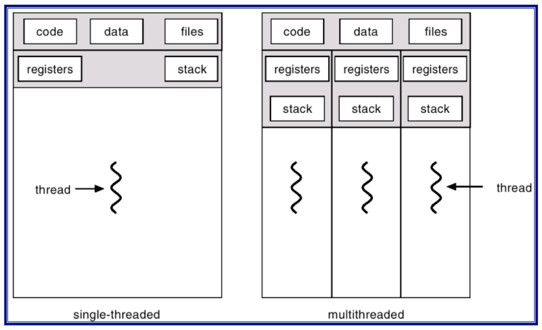
UNIX Process Control

- UNIX provides a number of system and library calls for process control including:
  - o fork - used to create a new process
  - o execve - to change the program a process is executing
  - o exit - used by a process to terminate itself normally
  - o abort - used by a process to terminate itself abnormally
  - o kill - used by one process to kill or signal another
  - o wait - to wait for termination of a child process
  - o sleep - suspend execution for a specified time interval
  - o getpid - get process id
  - o getppid - get parent process id

Why Threads?

- Multitasking OS can do more than one thing concurrently by running more than a single process
- Processes can do several things concurrently be running more than a single thread
- Each thread is a different stream of control that can execute its instructions independently.
- Ex: A program (e.g., Browser) may consist of the following threads:
  - GUI thread
  - I/O thread
  - computation

Single vs. Multi threaded

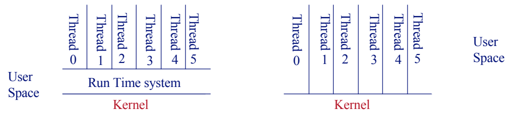


Processes vs. Threads

- A process includes
  - o An address space (code, data)
  - o A resource container (OS resource, accounting; e.g., file descriptors, memory map)
  - o A "thread of control", which defines where the process is currently executing (basically, PC, registers, and stack)
- Threads
  - o Separate the concepts of a "thread of control" from the rest of the process
  - o A thread has its own
    - Stack Pointer (SP), Program Counter (PC)
    - All the other resources are shared by **all** threads of that process, including open files, virtual address space, child processes

Two Approaches to Threads

- o User-level threads
- o Kernel-support threads



User vs. Kernel Threads

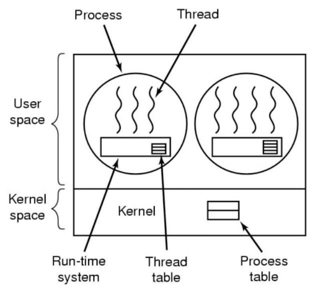
- Thread management done by user-level library
- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

- Supported by the kernel
- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - Linux

User vs. Kernel-Level Threads

- Question
  - What is the difference between user-level and kernel-level threads?
- Discussions
  - When a user-level thread is blocked on an I/O event, the whole process is blocked
  - A context switch of kernel-threads is expensive
  - A smart scheduler (two-level) can avoid both drawbacks

Implementing Threads in User Space

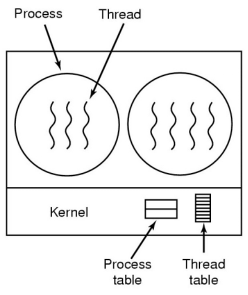


A user-level threads package

User-Level Threads

- The run-time support system for threads is entirely in user space.
- The threads run on top of a run-time system, which is a collection of procedures that manage threads.
- As far as the OS is concerned, it is a single (threaded) process.
- Threads can be implemented on an OS that does not support thread (as a library package in user space to do threads management)
- Each process can have its own customized scheduling algorithm.

Implementing Threads in the Kernel



A threads package managed by the kernel

Kernel-supported Threads

- No run-time system is needed.
- For each process, the kernel has a table with one entry per thread, for thread's registers, state, priority, and other information.
- All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure.
- When a thread blocks, the kernel can run either another thread from the same process, or a thread from a different process.

User-level vs. Kernel-supported Threads

- Assume:
  - Process A has one thread and Process B has 100 threads.
  - Each process receives the same number of time slices.
- User-level Thread:
  - A thread in process A runs 100 times as fast as a thread in process B.
  - One blocking system call blocks all threads in process B.
- Kernel-supported Threads:
  - Process B receives 100 times the CPU time than process A.
  - Switching among the thread is more time-consuming because the kernel must do the switch.
  - Process B could have 100 system calls in operation concurrently.

Processes and Threads

- A UNIX Process is
  - a running program with
  - a bundle of resources (file descriptor table, address space)
- A thread has its own
  - stack
  - program counter (PC)
  - All the other resources are shared by **all** threads of that process. These include:
    - open files
    - virtual address space
    - child processes

Typical Thread API

- Creation
  - Create, Join
- Mutual exclusion
  - Acquire (lock), Release (unlock)
- Condition variables
  - Wait, Signal, Broadcast

POSIX Threads

- IEEE POSIX created a standard for threads to ease porting of threaded applications
- Commonly called “pthreads”
- User-level thread implementation
- Other thread APIs (i.e., Solaris threads, Linux threads) use similar semantics

Thread Creation

- POSIX standard API for multi-threaded programming
- A thread can be created by **pthread_create** call
- pthread_create (&thread, 0, start, args)

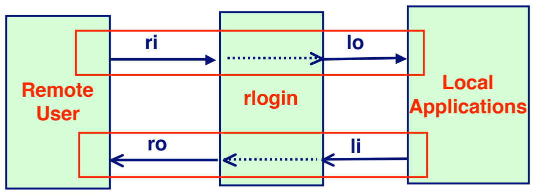
ID of new thread is returned in this variable

used to define thread attributes (e.g., Stack size)
0 means use default attributes

Name/address of the routine
where new thread should begin executing

Arguments passed to start

Example: When are threads useful?



Sample Code

```
typedef struct { int i, o } pair;
rlogind ( int ri, ro, li, lo) {
  pthread_t in_th, out_th;
  pair in={ri,lo}, out={li,ro};
  pthread_create(&in_th,0, incoming, &in);
  pthread_create(&out_th,0, outgoing, &out);
}
Note: 2 arguments are packed in a structure
```

Problem: If main thread terminates, memory for in and out structures may disappear, and spawned threads may access incorrect memory locations

If the process containing the main thread terminates, then all threads are automatically terminated, leaving their jobs unfinished.

Ensuring main thread waits…

```
typedef struct { int i, o } pair;
rlogind ( int ri, ro, li, lo) {
  pthread_t in_th, out_th;
  pair in={ri,lo}, out={li,ro};
  pthread_create(&in_th,0, incoming, &in);
  pthread_create(&out_th,0, outgoing, &out);
  pthread_join(in_th,0);
  pthread_join(out_th,0);
}
```

Thread Termination

- A thread can terminate
  - by executing **pthread_exit**, or
  - By returning from the initial routine (the one specified at the time of creation)
- Termination of a thread unblocks any other thread that's waiting using **pthread_join**
- Termination of a process terminates all its threads

Example: Hello World

```
void *print_message(void *ptr);

void main (void)
{
  pthread_t thread1, thread2;
  /* create first thread */
  if (pthread_create(&thread1, NULL, print_message, (void*) "Hello")) {
    perror("pthread_create : can't create thread 1");
    exit(-1);
  }
  /* create second thread */
  if (pthread_create(&thread2, print_message, (void*) "World")) {
    perror("pthread_create : can't create thread 2");
    exit(-2);
  }
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  exit(0);
}
```

Example: Hello World (continued)

Here's the function that gets executed by each thread:

```
void *print_message(void *ptr)
{
  char *msg = (char *) ptr;

  printf("%s ", msg);
  return NULL;
}
```

What's the output of the program?

- There is **no unique** answer since the output depends on how the thread scheduler schedules the individual threads.

Synchronization issues

- the threads execute concurrently
- there is no guarantee that the first thread reaches its `printf` statement prior to the second thread
- without the `pthread_join`, the parent thread *may* execute the `exit` statement prior to either of the threads reaching its `printf` statement.

  This would terminate the entire process including the two child threads without anything being printed!

✦ Never rely on the timing of the thread scheduler!

Thread Synchronization with Semaphores

```
int sem_init(sem_t *sem, int pshared,
             unsigned int value)
```

`sem_init` initializes the semaphore pointed to by `sem` to `count`.

if **`pshared`** is nonzero, then the semaphore is shared between processes.

Semaphores operations (continued)

- `int sem_wait(sem_t *sem)`
  blocks until the semaphore count pointed to by `sem` is greater than zero and then atomically decrements the count.
- `int sem_trywait(sema_t *sem)`
  atomically decrements the count pointed to by `sem`, if the count is greater than zero

What the difference?
With `sem_wait` if the lock is already zero, the process will block until it becomes zero. `sem_trywait` will return if the semaphore is already locked

Semaphores operations (continued)

- `int sem_post(sem_t *sem)`
  atomically increments the count of the semaphore pointed to by `sem`. If there are any threads blocked on the semaphore, one will be unblocked.
- `int sem_destroy(sema_t *sem)`
  destroys any state associated with the semaphore pointed to by `sem`.

Improved Hello World program

```
void *print_message(void *ptr);
sem_t world, barrier;

void main (void)
{
 pthread_t  thread1, thread2;
 sem_init(&world,   1, NULL, NULL);
 sem_init(&barrier, 0, NULL, NULL);

 if (pthread_create(&thread1, NULL, print_message, (void*) "Hello")) {
   perror("thread_create : can't create thread 1");
   exit(-1);
 }
 sem_wait(&world);

 /* to be continued ... */
```

Improved Hello World (continued)

```
 /* ... continued */
 if (thr_create(NULL, 0, print_message, (void*) "World", 0L, &thread2))
   exit(-2);
 sema_wait(&barrier);
 sema_wait(&barrier);      /* this ensures that both threads have printed */
 exit(0);
}
void *print_message(void *ptr)
{
 char *msg = (char *) ptr;
 printf("%s ", msg);
 sema_post(&world);
 sema_post(&barrier);
 return NULL;
}
```

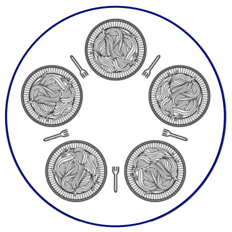
Deadlock

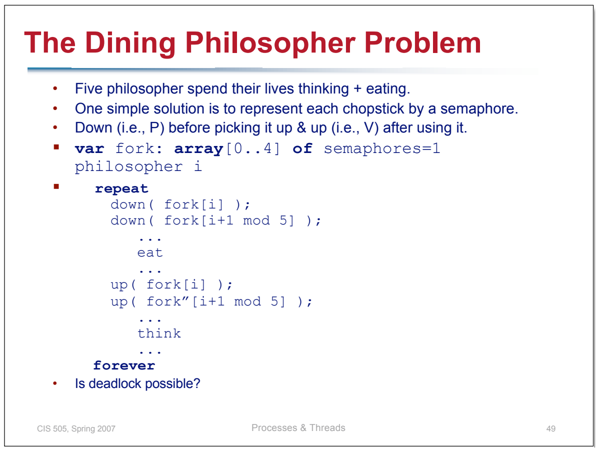
When programming with threads we have to be careful to avoid deadlock

```
Thread 1                      Thread 2
sem_wait(&s1);                sem_wait(&s2);
   sem_wait(&s2);                sem_wait(&s1);
   /* critical section */        /* critical section */
   sem_post(&s2);                sem_post(&s1);
sem_post(&s1);                sem_post(&s2);
```

Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock





The Dining Philosopher Problem

- Five philosopher spend their lives thinking + eating.
- One simple solution is to represent each chopstick by a semaphore.
- Down (i.e., P) before picking it up & up (i.e., V) after using it.

```
var fork: array[0..4] of semaphores=1
philosopher i
  repeat
    down( fork[i] );
    down( fork[i+1 mod 5] );
       ...
       eat
       ...
    up( fork[i] );
    up( fork"[i+1 mod 5] );
       ...
       think
       ...
  forever
```

- Is deadlock possible?