Chapter 2

Processes and Threads

2.1 The Process

- **Multiple activities happening**
  - **Multiprogramming**
  - **Multiprocessing**
- **Need conceptual model to handle "parallel" activities**
  - **Pseudoparallelism – giving the illusion of parallelism**
  - **Multiprocessor – true hardware parallelism**

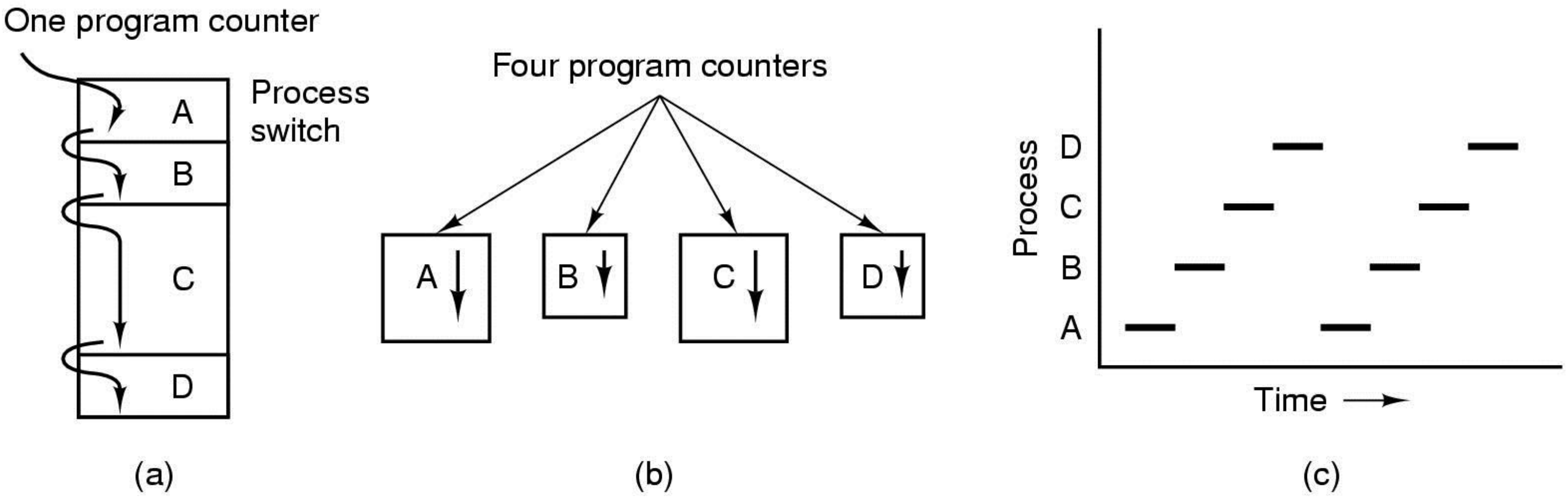


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

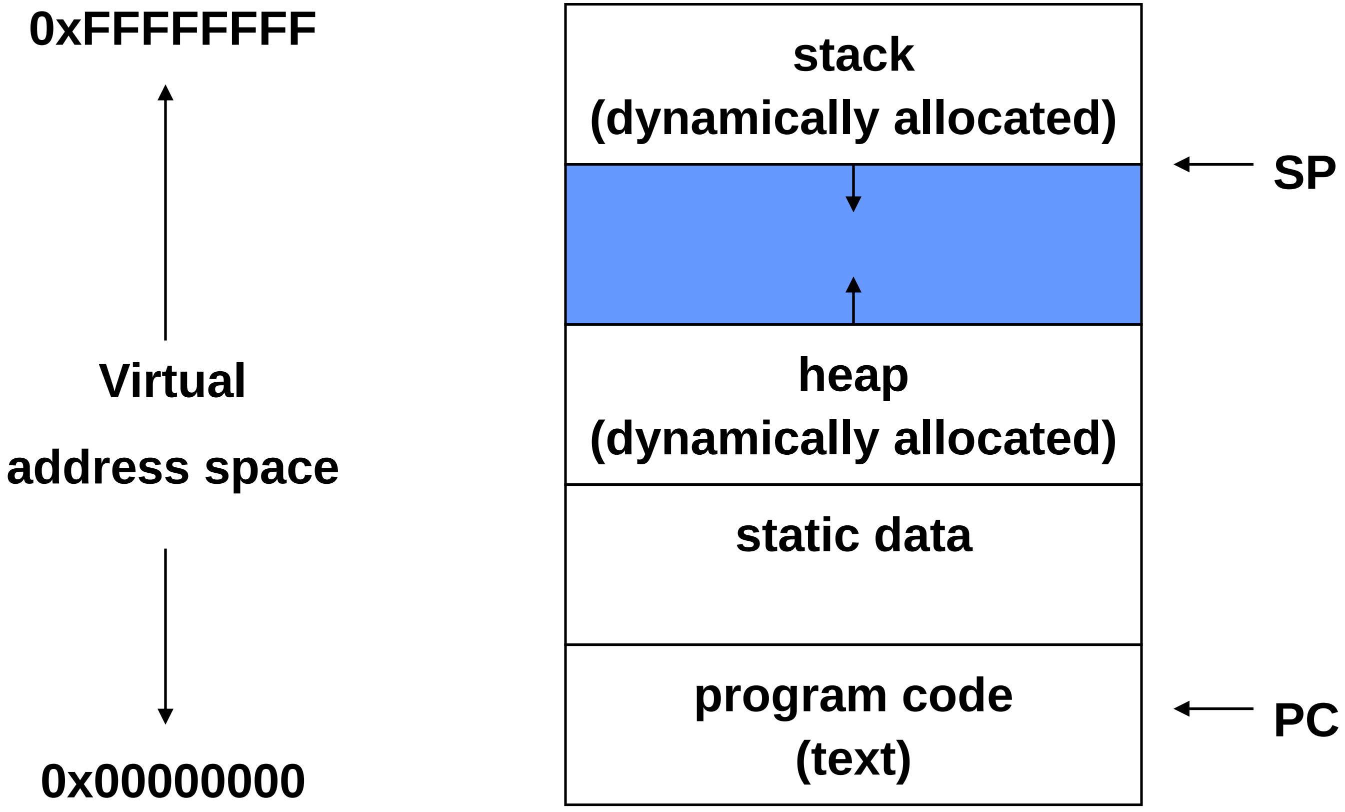
What is a process?

- The process is the OS’s abstraction for execution
  - the unit of execution
  - the unit of scheduling
  - the dynamic (active) execution context
    - compared with program: static, just a bunch of bytes
- Process is often called a job, task, or sequential process
  - a sequential process is a program in execution
    - defines the instruction-at-a-time execution of a program
- Difference between process and program?

Process

- More than a program, a process has
  - an *address space* – usually protected and virtual – mapped into memory
  - the *code* for the running program
  - The *data* for the running program
  - an *execution stack* and *stack pointer* (SP); also *heap*
  - the *program counter* (PC)
  - a set of processor *registers* – general purpose and status
  - a set of system *resources*
    - files, network connections, pipes, …
    - privileges, (human) user association, …
  - **And more…**

Process Address Space (traditional Unix)



Processes in the OS – Representation

- To users (and other processes) a process is identified by its *Process ID* (PID)

- In the OS, processes are represented by entries in a *Process Table* (PT)
  - PID is index to (or pointer to) a PT entry
  - PT entry = *Process Control Block* (PCB)

- PCB is a large data structure that contains or points to all info about the process
  - Linux – defined in **`task_struct`** (over 70 fields)
    - see **`include/linux/sched.h`**
  - Windows XP – defined in *EPROCESS* – about 60 fields

Process creation

- One process can create another process
  - creator is called the parent
  - created process is called the child
  - UNIX: do `ps`, look for PPID field
  - what creates the first process, and when?
- In some systems, parent defines or donates resources and privileges for its children
  - UNIX: child inherits parents userID field, etc.
- when child is created, parent may either wait for it to finish, or it may continue in parallel, or both!
- **Principal events that cause process creation**
  - **System Initialization, Execution of a process creation system call, User request to create a new process, Initiation of a batch job**

Process Creation: `fork()`

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, -1 on error

- **The only way a new process is created by the Unix kernel**
- **Child process: the new process created by fork()**
- **fork() called once, but returns *twice*!**
  - **0 in child and process ID of child in parent process**
- **The child copies the parent, such as data, heap, stack and DO NOT share these portions of memory (different address space)**

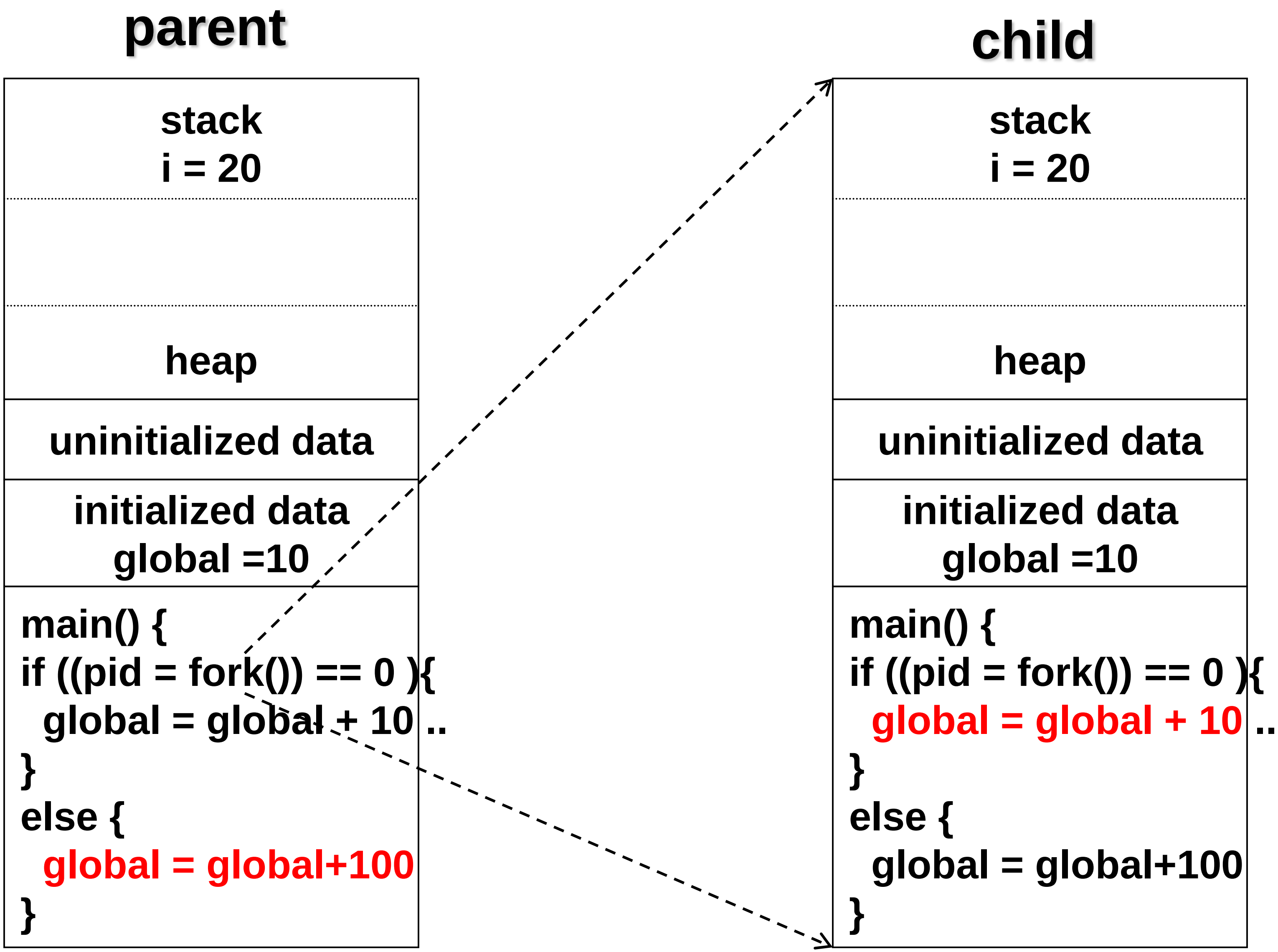
fork()

```
int global = 10;
main() {
  int i = 20; pid_t pid; int status;

  if ((pid = fork()) == 0) { /* child process */

    global = global+10; i = i + 10;
  }
  else {
    /* fork() returns a pid != 0 */
    /* parent process            */]

    global = global+100; i = i + 100;

    wait(&status); /* or wait(); */

  }

  printf("global = %d; i = %d\n",global,i);
}
```

```
$a.out
global = 110; i = 120
global = 20; i = 30
```

```
$a.out
global = 20; i = 30
global = 110; i = 120
```

What happens with fork()



parent

stack
i = 20

heap

uninitialized data

initialized data
global =10

```
main() {
if ((pid = fork()) == 0 ){
  global = global + 10 ..
}
else {
  global = global+100
}
```

child

stack
i = 20

heap

uninitialized data

initialized data
global =10

```
main() {
if ((pid = fork()) == 0 ){
  global = global + 10 ..
}
else {
  global = global+100
}
```

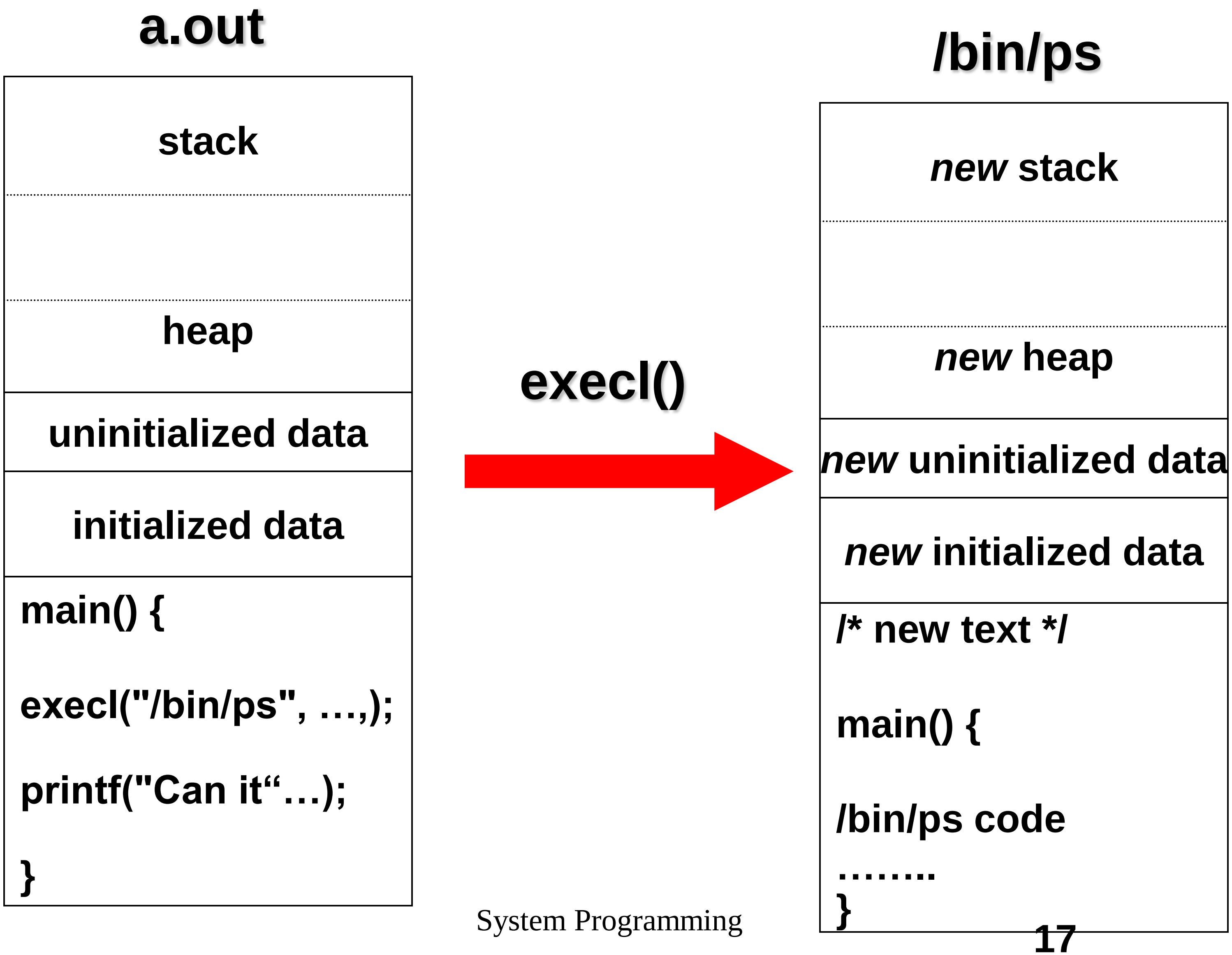
Program Execution: exec functions

- The process calling exec is *completely replaced by the new program* (text, data, heap, and stack) and the new program starts at its main().

```
int main() {

  execl("/bin/ps", "put filename here", "a", (char*) 0);

  printf("Can it print this message? ERROR!\n");
}
```

```
$ a.out
  PID TTY       STAT   TIME COMMAND
14886 pts/3     S      0:00 -bash
14911 pts/3     T      0:00 emacs -nw exec.c
15080 pts/3     R      0:00 put filename here a
```

Program Execution: `exec` functions



a.out

stack

heap

uninitialized data

initialized data

main() {

execl("/bin/ps", …,);

printf("Can it“…);

}

execl()

/bin/ps

new stack

new heap

new uninitialized data

new initialized data

/* new text */

main() {

/bin/ps code

……..

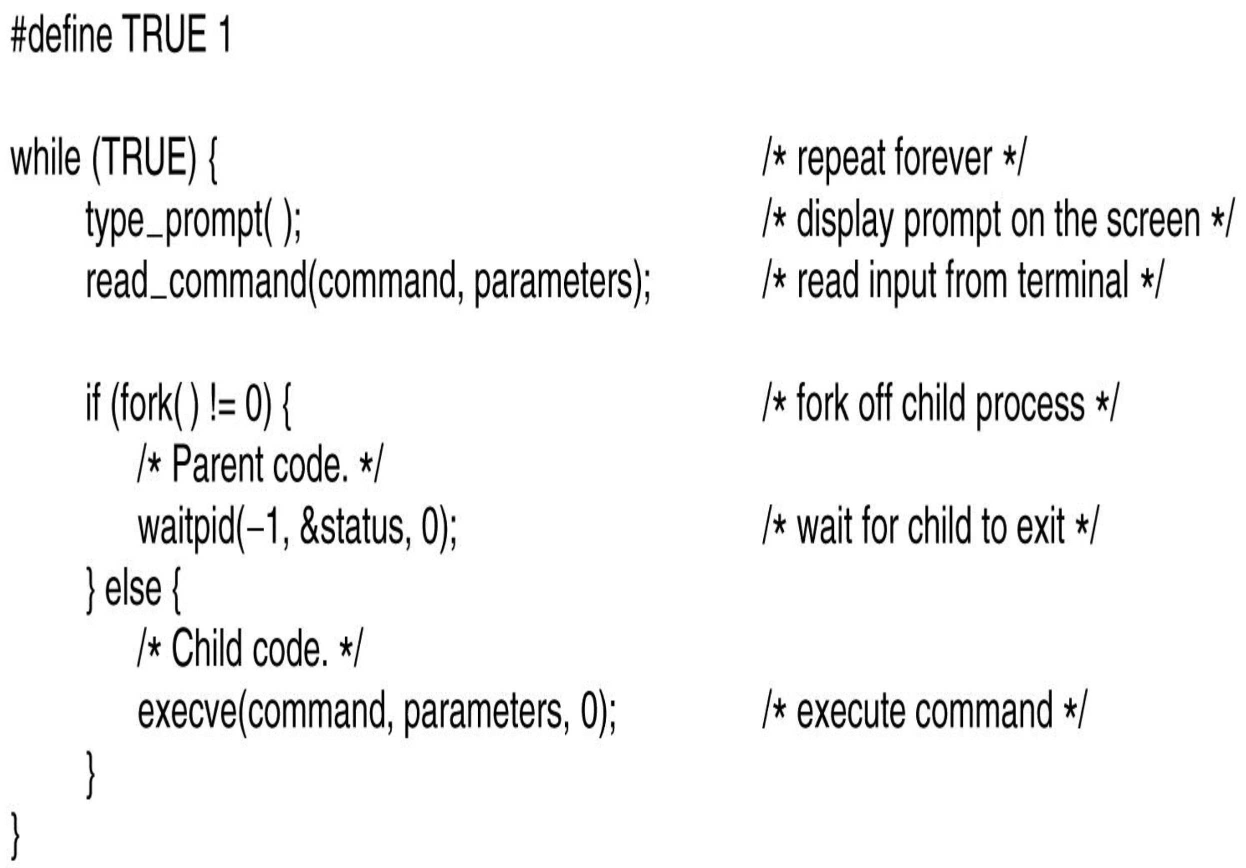
}

exec functions

```
#include <unistd.h>
int execl(const char *pathname,const char *arg0, … /* (char*) 0 */);
int execv(const char *pathname,const char *argv[]);
int execle(const char *pathname,const char *arg0, …
                                   /* (char*) 0, char *const envp[] */);
int execve(const char *pathname,const char *argv[],char *const envp[]);
int execlp(const char *filename,const char *arg0, … /* (char*) 0 */);
int execvp(const char *filename,const char *argv[]);
```

- **Six functions: postfix meaning (See Fig 8.5 and 8.6)**
  - **`p`: takes a *filename* argument, uses the `PATH` environment variable.**
  - **`l`: takes a list of arguments. (the last argument should be a null pointer *(char *) 0*)**
  - **`v`: takes *argv[]* vector.**
  - **`e`: takes *envp[]* array. (without `e`, the environment variables of the calling process are copied)**
- **`execve` is the only system within the kernel.**
- **`fork()` and then `exec()`: `spawn()`**

A simple Shell



```
#define TRUE 1

while (TRUE) {                                    /* repeat forever */
      type_prompt( );                             /* display prompt on the screen */
      read_command(command, parameters);          /* read input from terminal */

      if (fork( ) != 0) {                         /* fork off child process */
          /* Parent code. */
          waitpid(−1, &status, 0);                /* wait for child to exit */
      } else {
          /* Child code. */
          execve(command, parameters, 0);         /* execute command */
      }
}
```

Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
   - Often due to a program bug
   - Illegal instructions, referencing nonexistent memory, dividing by zero, …
4. Killed by another process (involuntary)
   - Kill()

Process Termination: `exit()`

- **Normal termination**
  - **return from the `main()`**
  - **Calling `exit()` (defined by ANSI C)**
    - calls all exit handlers registered by `atexit()`
    - close all standard I/O streams
  - **Calling `_exit()`**
    - called by exit() and handles the Unix-specific details
- **Abnormal termination**
  - **Calling `abort()` - generates `SIGABRT` signal**
  - **The reception of certain *signals***

- **Regardless how a process terminates, the same code in the kernel is eventually executed; *it closes all the open descriptors for the process, release memory, and so on.***

wait() and waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
                Both returns: process ID if OK, 0, or -1 on error
```

- When a process terminates, the kernel sends the parent the `SIGCHLD` signal (*it is an asynchronous event*).
- A process that calls wait functions can
  - block *(if all of its children are still running)*, or
  - return immediately with the termination status of a child, or
  - return immediately with an error *(if it doesn't have any children)*.
- ***statloc***
  - A pointer to an integer
  - stores the termination status of the terminated process

wait()and waitpid()

```
int main(void){   pid_t pid; int      status;
   if ( (pid = fork()) < 0) err_sys("fork error");
   else if (pid == 0) exit(7);   /* child */

   if (wait(&status) == pid)     /* wait for child */
      pr_exit(status);           /* and print its status */


   if ( (pid = fork()) < 0) err_sys("fork error");
   else if (pid == 0) abort(); /* child : generates SIGABRT */

   if (wait(&status) == pid)     /* wait for child */
       pr_exit(status);          /* and print its status */


   if ( (pid = fork()) < 0) err_sys("fork error");
   else if (pid == 0) status /= 0; /* child: divide by 0 generates
                                    SIGFPE */
   if (wait(&status) == pid)     /* wait for child */
       pr_exit(status);           /* and print its status */
}
```

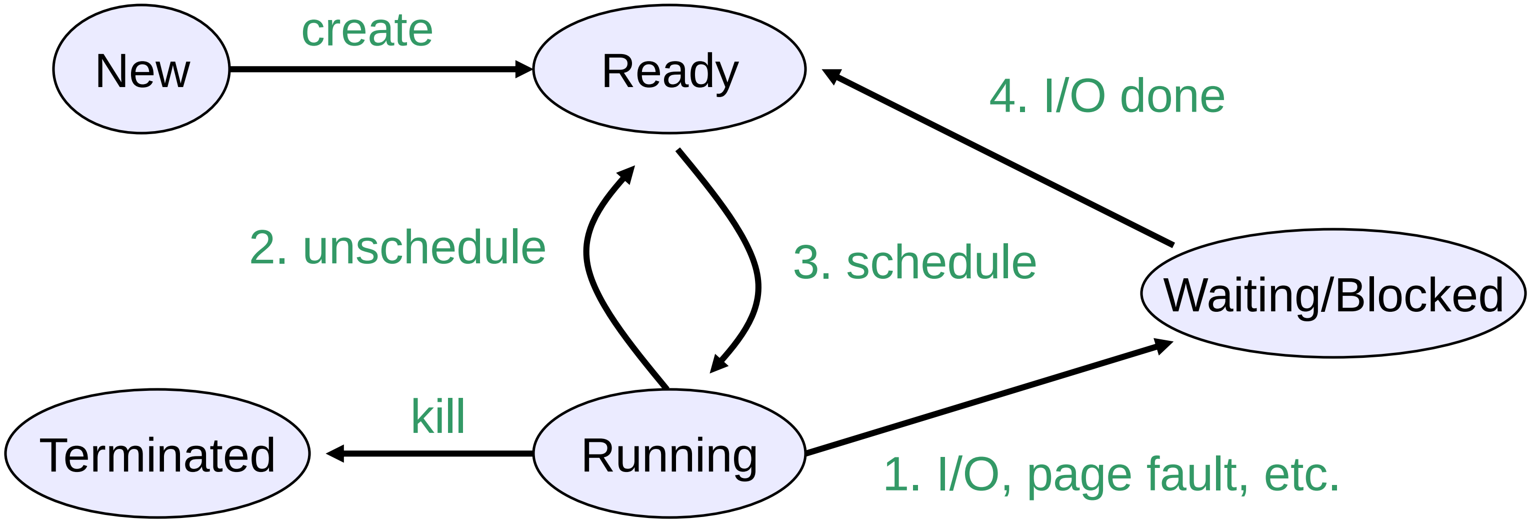
2.1.4 Process Hierarchy

- Parent-child relation
  - Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group“
  - A signal can be sent to a process group
- Windows has no concept of process hierarchy
  - all processes are created equal
  - Processes can give away their children

2.1.5 Process states

- **Each process has an execution state, which indicates what it is currently doing**
  - **ready: waiting to be assigned to CPU**
    - could run, but another process has the CPU
  - **running: executing on the CPU**
    - is the process that currently controls the CPU
    - how many processes can be running simultaneously?
  - **blocked: waiting for an event, e.g. I/O**
    - cannot make progress until event happens
- **As a process executes, it moves from state to state**
  - **UNIX: run ps, STAT column shows current state**
  - **which state is a process in most of the time?**

Process state transitions



- **What can cause schedule/unschedule transitions?**
- **A preemptive scheduler will force a transition from running to ready. A non-preemptive scheduler waits.**

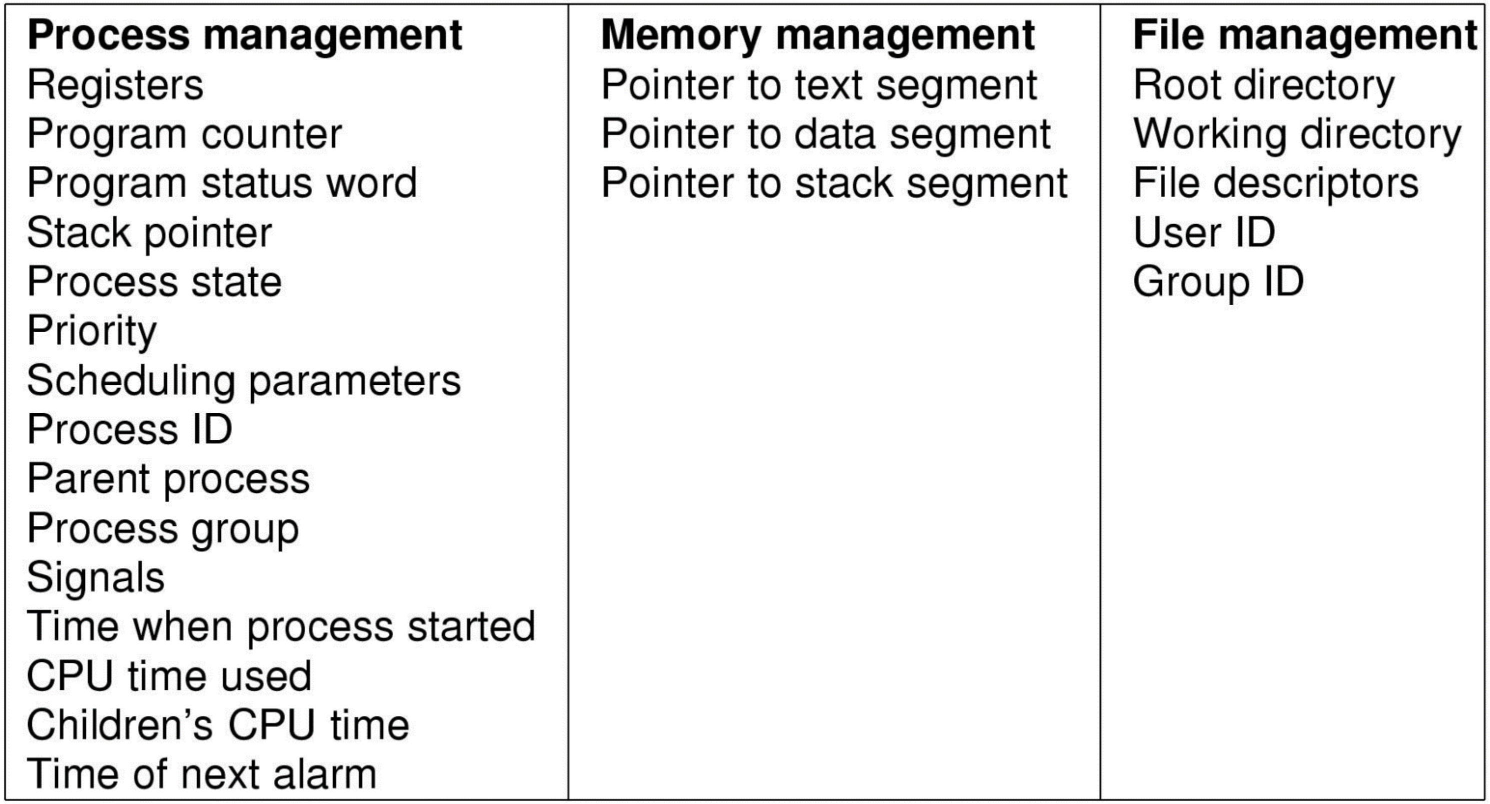
2.1.6 Implementation of Processes

- **A process consists of (at least):**
  - **an address space**
  - **the code for the running program**
  - **the data for the running program**
  - **an execution stack and stack pointer (SP)**
    - traces state of procedure calls made
  - **the program counter (PC), indicating the next instruction**
  - **a set of general-purpose processor registers and their values**
  - **a set of OS resources**
    - open files, network connections, sound channels, …
- **The process is a container for all of this state**
  - **a process is named by a process ID (PID)**
    - just an integer (actually, typically a short)

Process data structures

- **How does the OS represent a process in the kernel?**
  - **at any time, there are many processes, each in its own particular state**
  - **the OS data structure that represents each is called the process control block (PCB) or process table**
- **PCB contains all info about the process**
  - **OS keeps all of a process' hardware execution state in the PCB when the process isn't running**
    - PC
    - SP
    - registers
  - **when process is unscheduled, the state is transferred out of the hardware into the PCB**

PCB, again



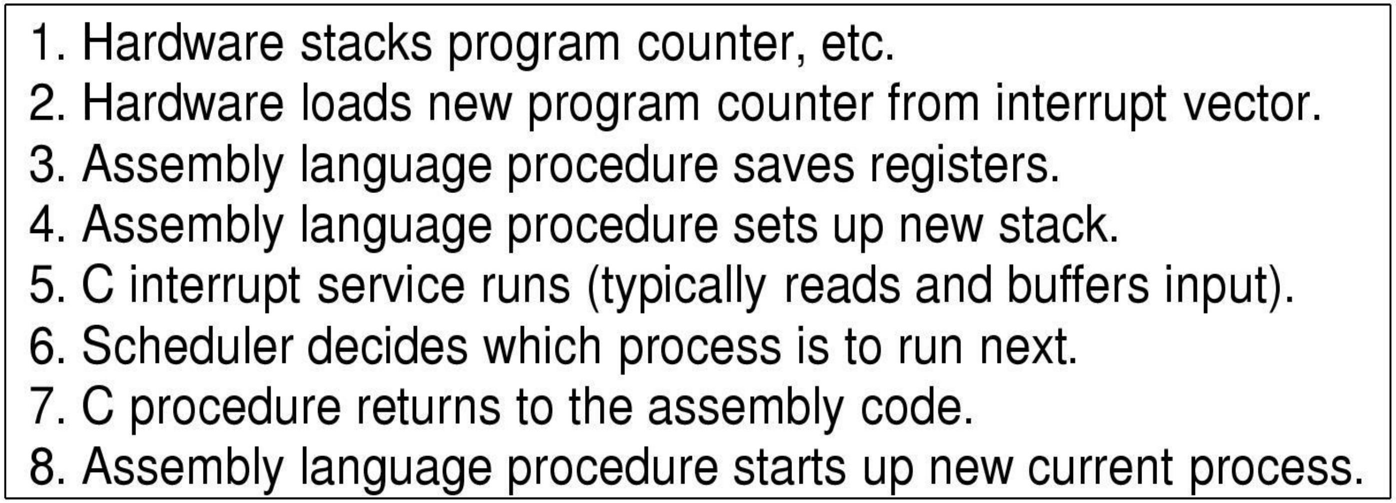
Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Fields of a process table entry

PCBs and Hardware State

- When a process is running, its hardware state is inside the CPU
  - PC, SP, registers
  - CPU contains current values
- When the OS stops running a process (puts it in the waiting state), it saves the registers' values in the PCB
  - when the OS puts the process in the running state, it loads the hardware registers from the values in that process' PCB
- The act of switching the CPU from one process to another is called a context switch
  - timesharing systems may do 100s or 1000s of switches/s
  - takes about 5 microseconds on today's hardware

Example of how a context switch occurs

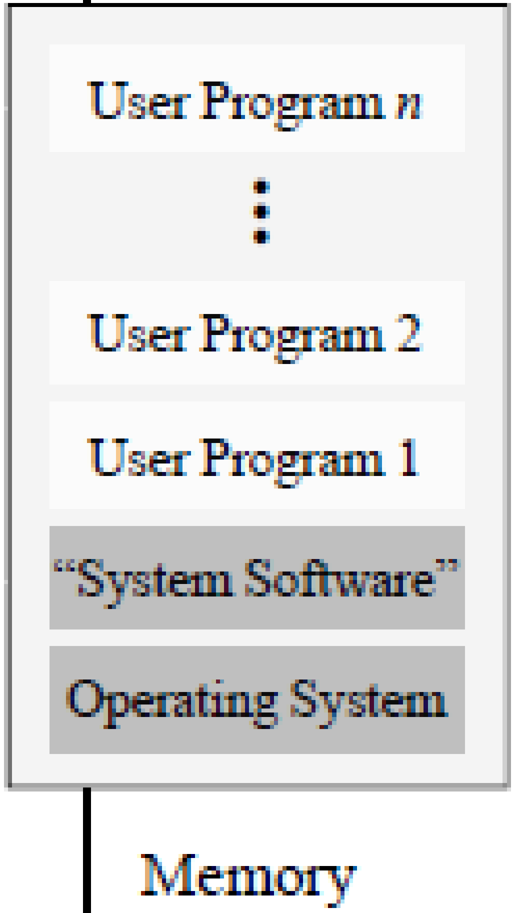


1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what lowest level of OS does when an interrupt occurs

Process Contexts

Example: Multiprogramming



User Program *n*

⋮

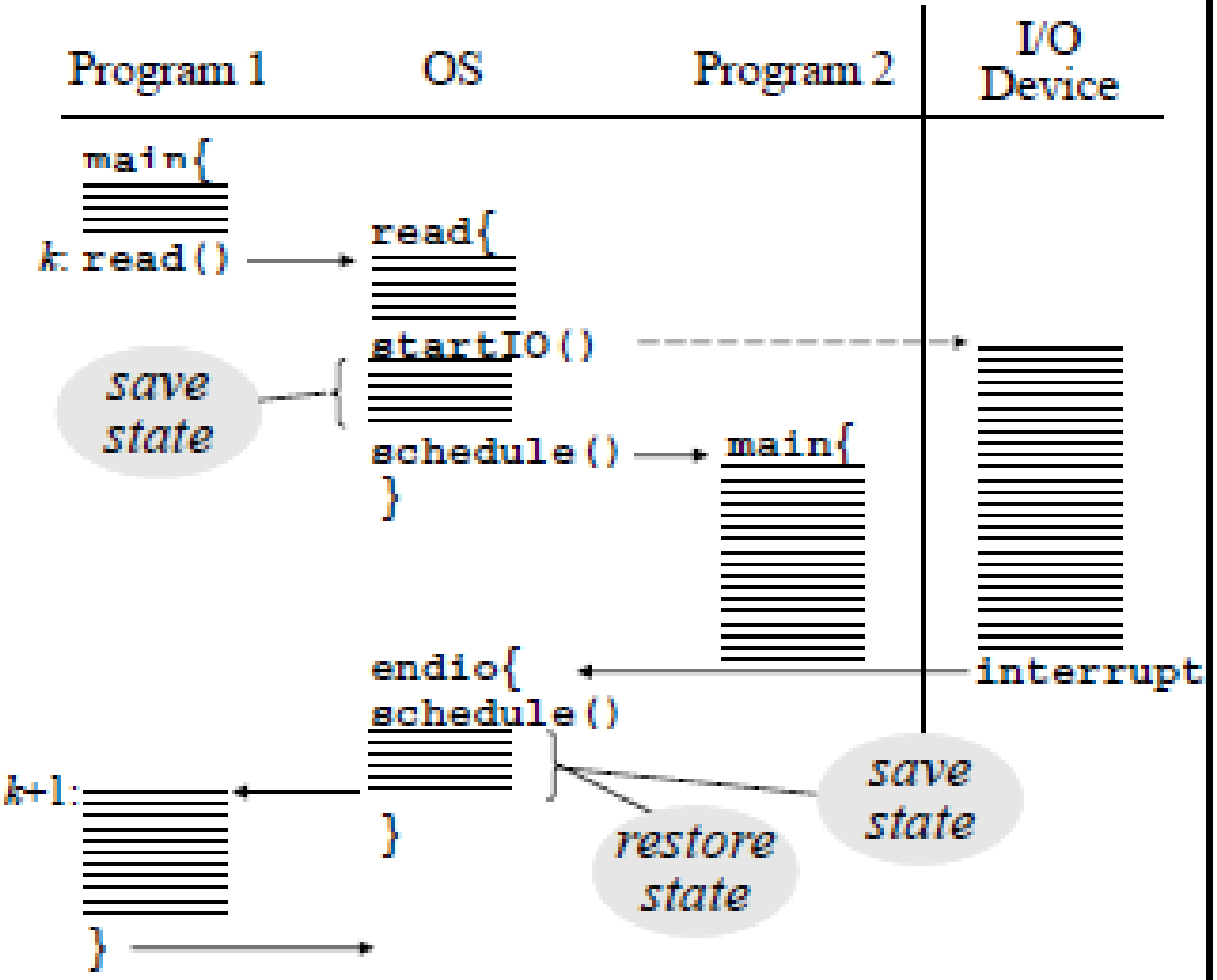
User Program 2

User Program 1

"System Software"

Operating System

Memory



Program 1

OS

Program 2

I/O Device

```
main{
k: read()
read{
startIO()
schedule()
}
main{
endio{
schedule()
}
k+1:
}
interrupt
```

save state

save state

restore state

2.2 Something different; threads

- **A process includes many things:**
  - **an address space (all the code and data pages)**
    - protection boundary
  - **OS resources (e.g., open files) and accounting info**
  - **hardware execution state (PC, SP, regs)**

Parallel Programs

- **Imagine a web server, which forks off copies of itself to handle multiple simultaneous tasks**
  - **or, imagine we have any parallel program on a multiprocessor**
- **To execute these, we need to:**
  - **create several processes that execute in parallel**
  - **cause each to map to the *same* address space to share data**
    - see the `shmget()` system call for one way to do this (kind of)
  - **have the OS schedule them in parallel**
    - multiprogramming or true parallel processing on an SMP
- **This is really inefficient**
  - **space: PCB, page tables, etc.**
  - **time: creating OS structures, fork and copy addr space, etc.**

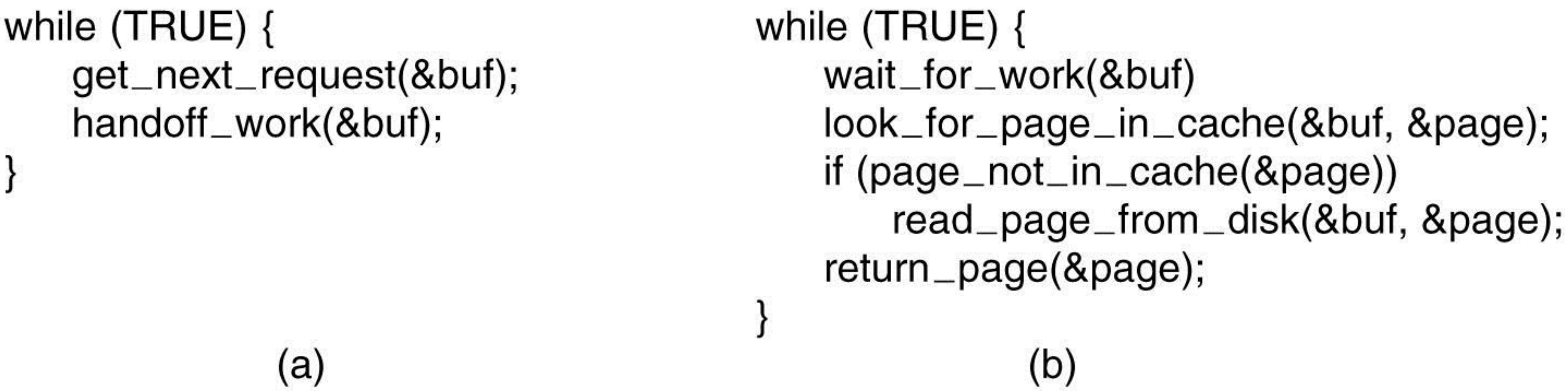
Can we do better?

- **What’s similar in these processes?**
  - **they all share the same code and data (address space)**
  - **they all share the same privileges**
  - **they all share the same resources (files, sockets, etc.)**
- **What’s different?**
  - **each has its own hardware execution state**
    - PC, registers, stack pointer, and stack
- **Key idea:**
  - **separate the concept of**
    - a process (address space, etc.) from that of
    - a minimal “thread of control” (execution state: PC, etc.)
    - i.e. process = resource group + thread of execution
  - **this execution state is usually called a thread, or sometimes, a lightweight process**

Can we do better with Threads

- Many applications have multiple activities going on at once
  - Thread programming model is simpler
  - Threads share an address space
- Threads are lighter weight than processes
  - Easier to create and destroy than process
- Performance gains
  - Overlapping I/O and CPU activities
  - Blocking system calls (I/O etc)
- Threads are useful on systems with multiple CPUs

Thread Usage (3)



```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.

Threads and processes

- **Most modern OS's (Mach, Chorus, NT, modern Unix) therefore support two entities:**
  - **the process, which defines the address space and general process attributes and resources(such as open files, pending alarms, signal handlers, accounting info etc.)**
  - **the thread, which defines a sequential execution stream within a process (program counter, registers, stack)**
- **A thread is bound to a single process**
  - **processes, however, can have multiple threads executing within them**
  - **sharing data between threads is cheap: all see same address space**
- **Threads become the unit of scheduling (or context switch unit)**
  - **processes are just containers in which threads execute**
  - **State transitions (running, blocked, ready, etc): same as processes.**

The Classical Thread Model (1)

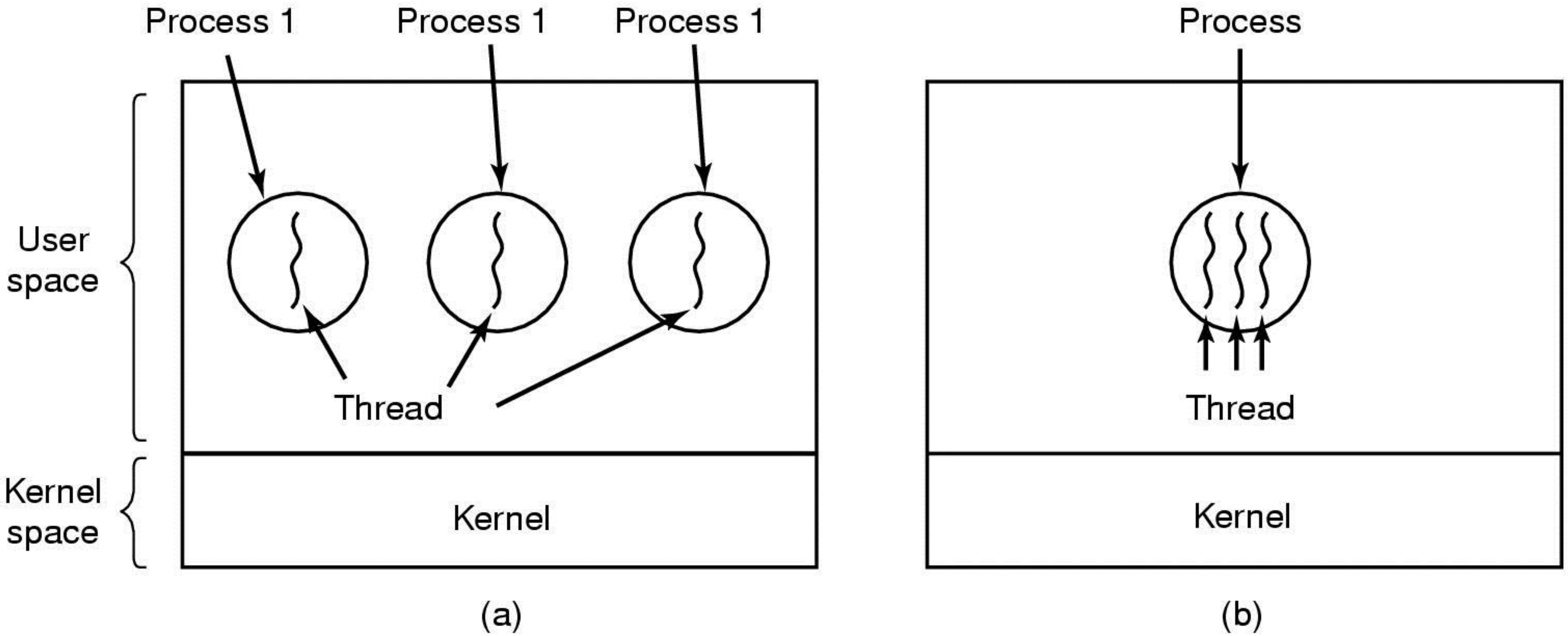
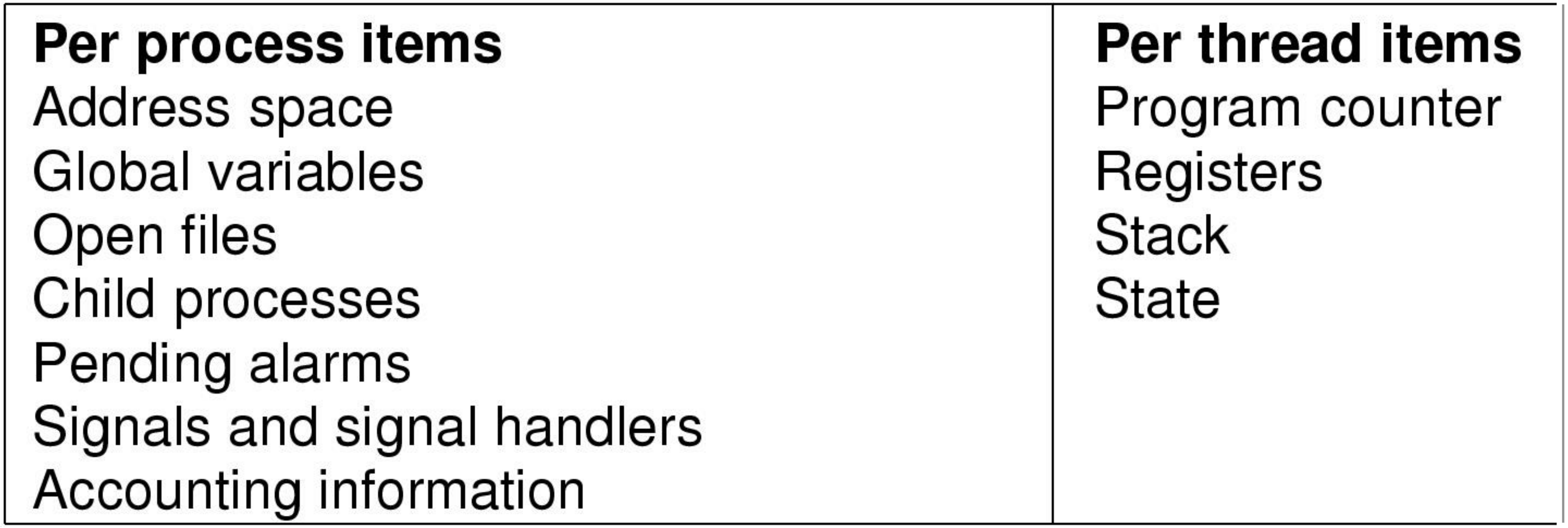


Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.

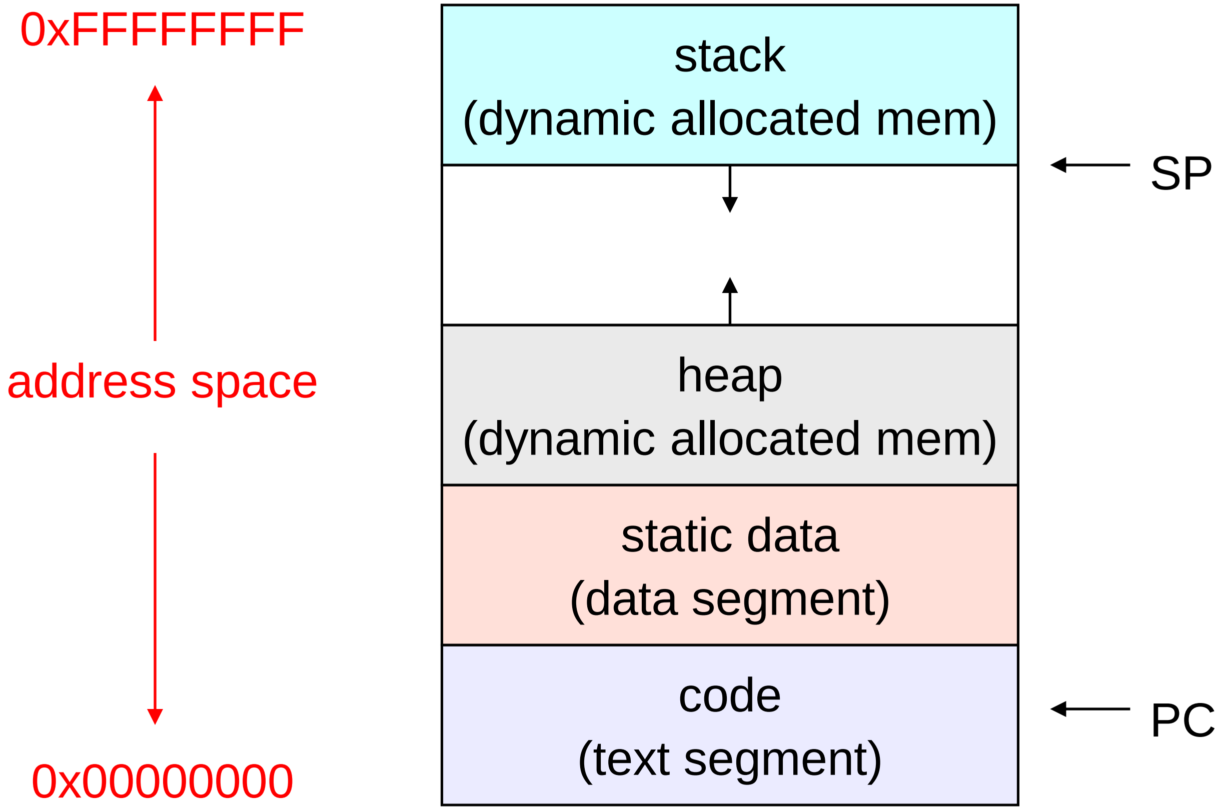
Threads and processes



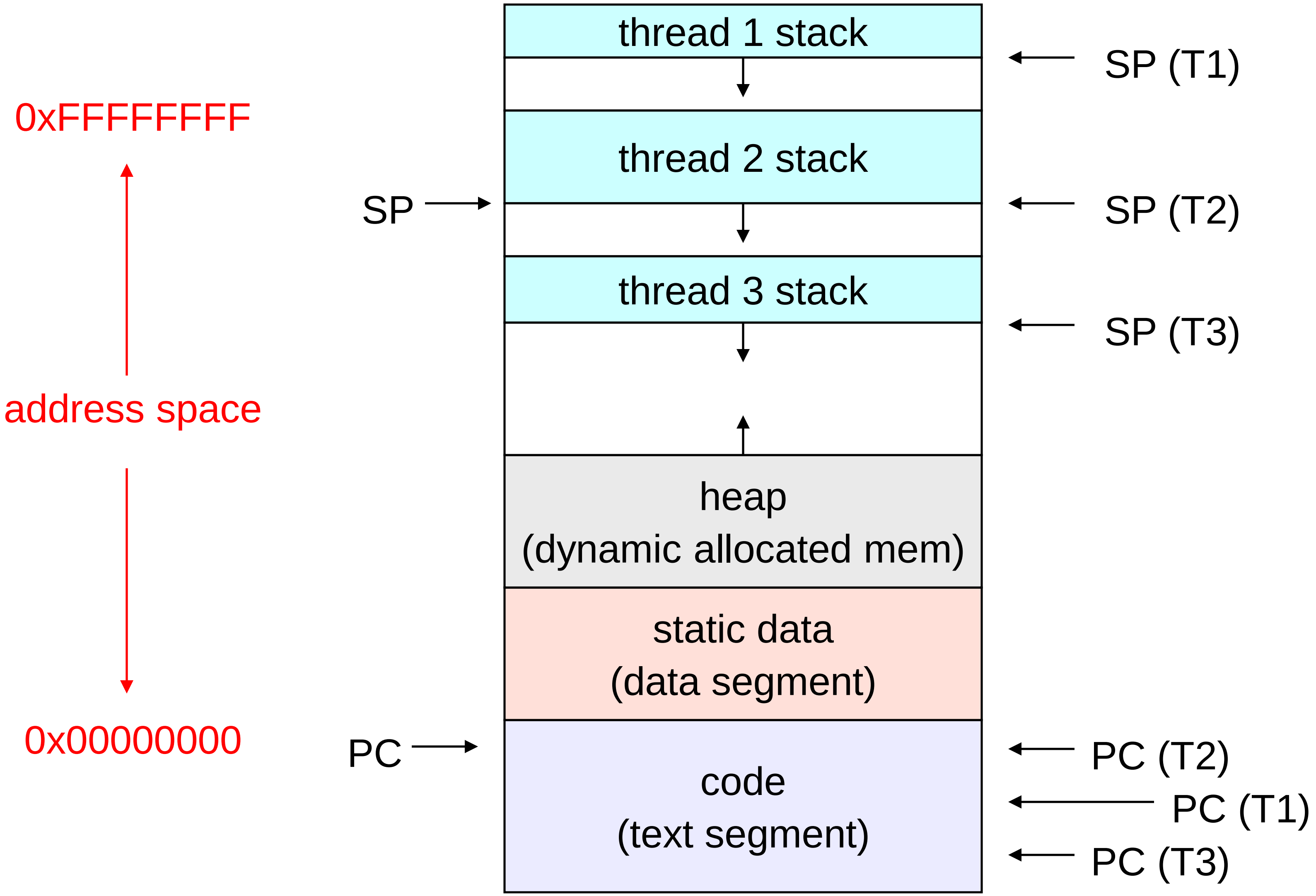
Per process items	Per thread items
Address space Global variables Open files Child processes Pending alarms Signals and signal handlers Accounting information	Program counter Registers Stack State

Items shared by all threads in a process and Items private to each thread

(old) Process address space



(new) Address space with threads



The Classical Thread Model (3)

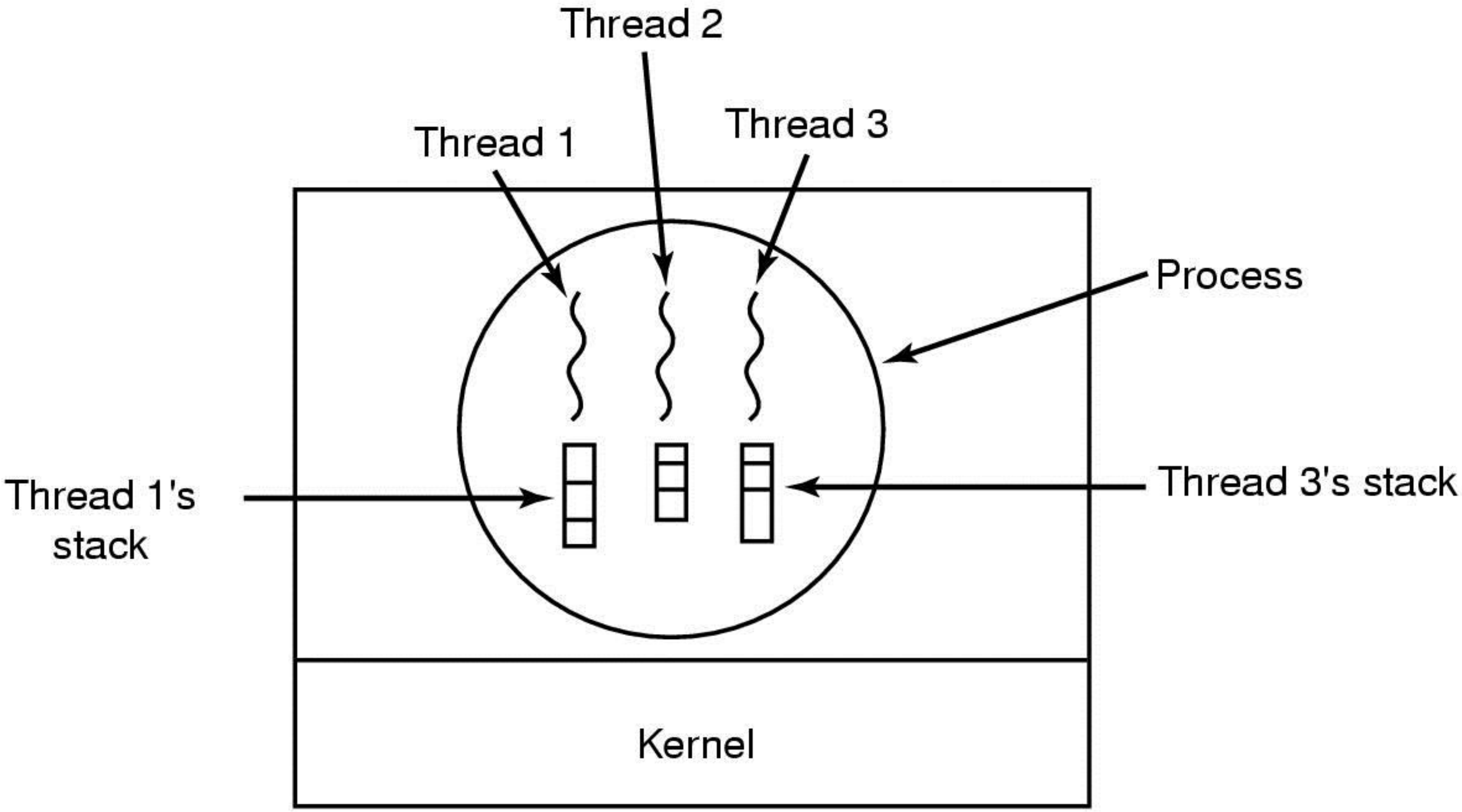
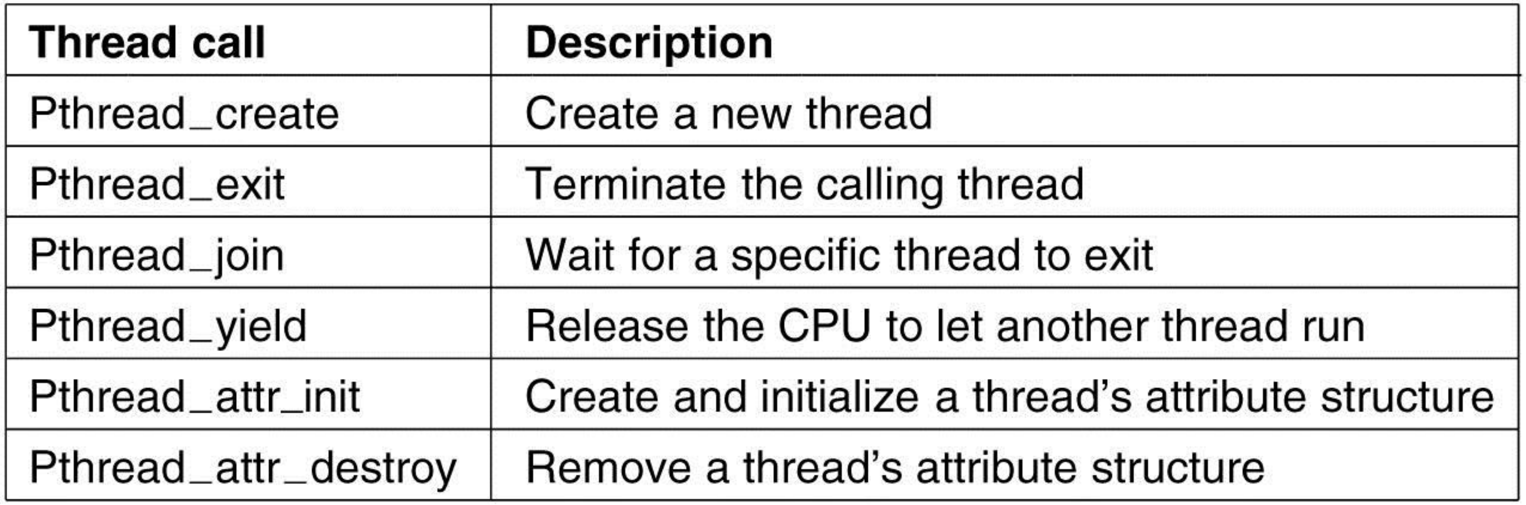


Figure 2-13. Each thread has its own stack.

Performance example

- **Creating a new process is costly, because of all of the data structures that must be allocated/initialized**
  - **Linux: over 95 fields in task_struct**
    - on a 700 MHz pentium, fork+exit = 251 microseconds, fork+exec = 1024 microseconds
- **Interprocess communication is costly, since it must usually go through the OS**
  - **overhead of system calls**
    - 0.46 microseconds on 700 MHz pentium
- **On a 700MHz Pentium running Linux 2.2.16:**
  - **Processes**
    - `fork/exit`: 251 μs
  - **Kernel threads**
    - `pthread_create()/pthread_join()`: 94 μs
  - **User-level threads**
    - `pthread_create()/pthread_join`: 4.5 μs

POSIX Threads (1)



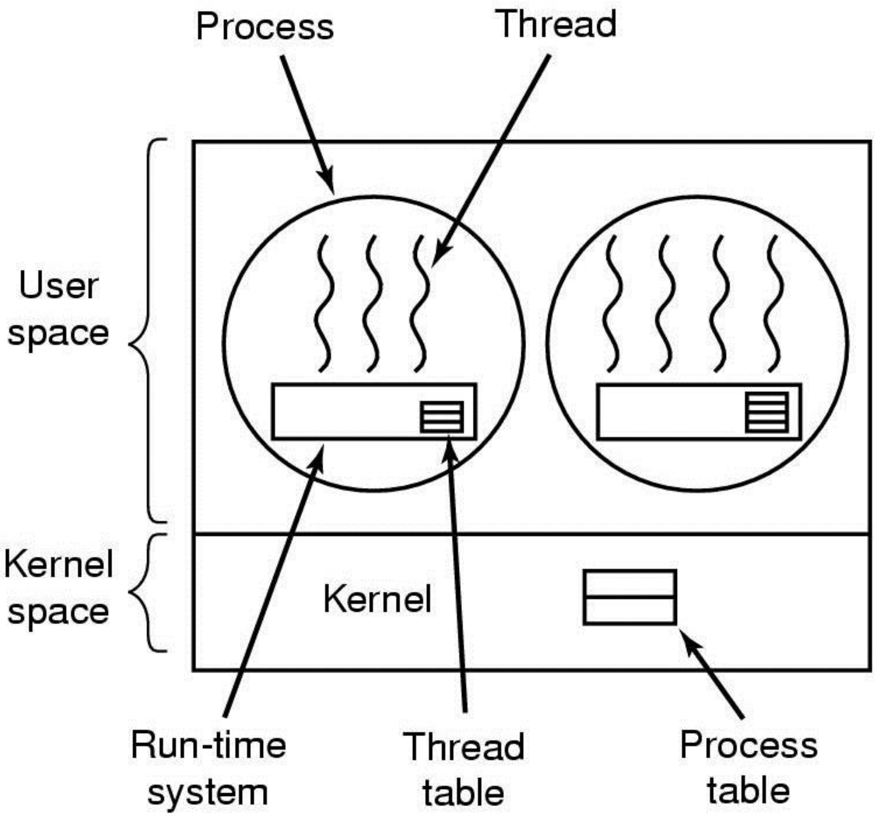
Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread’s attribute structure
Pthread_attr_destroy	Remove a thread’s attribute structure

Figure 2-14. Some of the Pthreads function calls.

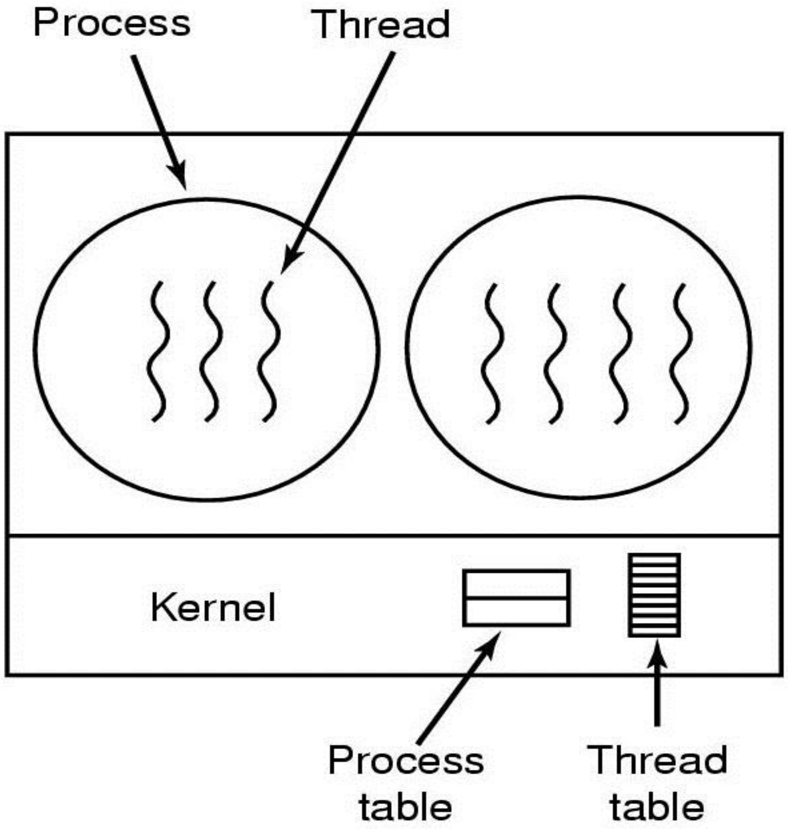
2.2.3 and beyond:
Kernel threads and user-level threads

- **Who is responsible for creating/managing threads?**
- **Two answers, in general:**
  - **the OS (kernel threads) (Section 2.2.5)**
    - thread creation and management requires system calls
  - **the user-level process (user-level threads) (Section 2.2.4)**
    - a library linked into the program manages the threads
- **Why is user-level thread management possible?**
  - **threads share the same address space**
    - therefore the thread manager doesn't need to manipulate address spaces
  - **threads only differ in hardware contexts (roughly)**
    - PC, SP, registers
    - these can be manipulated by the user-level process itself!

Kernel threads and user-level threads



A user-level threads package



A threads package managed by the kernel

User-Level Threads

- **To make threads cheap and fast, they need to be implemented at the user level**
  - **managed entirely by user-level library, e.g. `libpthreads.a`**
- **User-level threads are small and fast**
  - **Each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)**
  - **Each process needs its own private tread table to keep TCBs**
  - **creating a thread, switching between threads, and synchronizing threads are done via procedure calls**
    - no kernel involvement is necessary!
  - **user-level thread operations can be 10-100x faster than kernel threads as a result – no trap/context(process) switch/cache flush**
- **Can use a customized scheduling algorithm**
  - **A garbage collector thread does not stop in the middle of execution.**
- **Scale better than kernel threads**

User-level Thread Limitations

- **But, user-level threads aren’t perfect**
  - **tradeoff, as with everything else**

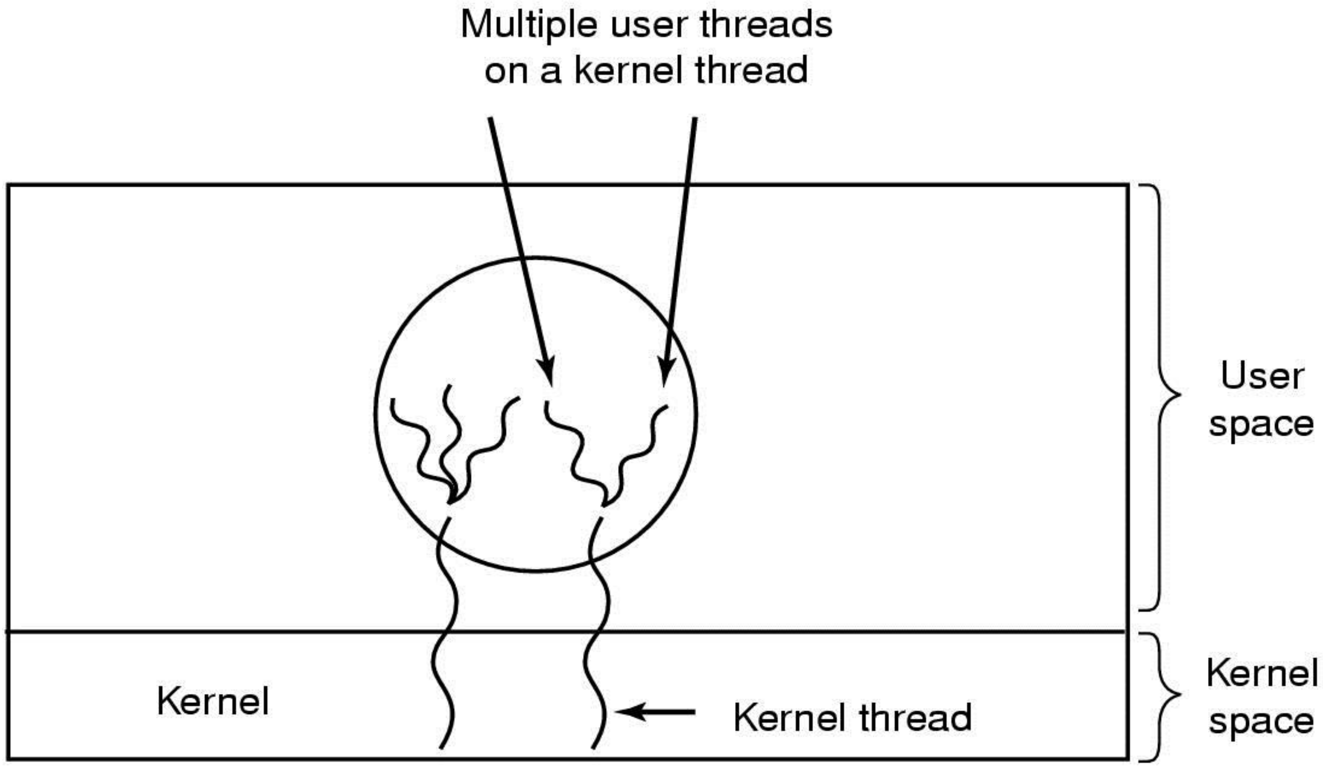
- **User-level threads are invisible to the OS**
  - **There is no integration with the OS**
  - **Blocking System Calls**
    - Use nonblocking calls
    - Check in advanced if the call will block : wrapper
      - Use select system call
  - **Page fault**
  - **A thread may run forever**
    - No clock interrupt
    - The scheduler can not get control back
    - Use a clock signal – crude and messy
  - **Non-concurrency of threads on multiple processors**
    - Threads in a process can not run on different processors

Kernel Threads

- **OS now manages threads *and* processes**
  - **All thread operations are implemented in the kernel**
  - **OS schedules all of the threads in a system**
    - if one thread in a process blocks (e.g. on I/O), the OS knows about it, and can run other threads from that process
    - possible to overlap I/O and computation inside a process
- **But, they can still be too expensive**
  - **thread operations are all system calls**
    - OS must perform all of the usual argument checks
    - but want them to be as fast as a procedure call!
  - **must maintain kernel state for each thread**
    - can place limit on # of simultaneous threads, typically ~1000
  - **A context switch of kernel threads is more expensive than user threads**

- **Creation and context switch is cheaper than processes'**

Hybrid Implementations



Multiplexing user-level threads onto kernel- level threads

- **Kernel is aware of only the kernel-level threads**

Scheduler Activations

- **Goal – mimic functionality of kernel threads**
  - gain performance of user space threads
- **Avoids unnecessary user/kernel transitions**
  - The user-space run-time system can block the synchronizing thread and schedule a new one
- **Kernel use upcall**
  - Notify the run-time system info on a blocked thread
  - Notify the run-time system when the blocked thread is ready again.
- **Problems:**
  - Fundamental reliance on kernel (lower layer)
  - calling procedures in user space (higher layer)

2.3 Synchronization (Interprocess Communication)

- **Threads cooperate in multithreaded programs**
  - **to share resources, access shared data structures**
    - e.g., threads accessing a memory cache in a web server
  - **also, to coordinate their execution**
    - e.g., a disk reader thread hands off a block to a network writer
- **For correctness, we have to control this cooperation**
  - **must assume threads interleave executions arbitrarily and at different rates**
    - scheduling is not under application writers' control
  - **we control cooperation using synchronization**
    - enables us to restrict the interleaving of executions
- **Note: this also applies to processes, not just threads**
  - **and it also applies across machines in a distributed system**

2.3.1 Race Condition

- *output is non-deterministic, depends on timing*
- **Suppose we have to implement a function to withdraw money from a bank account:**

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

- **Now suppose that you and your S.O. share a bank account with a balance of $100.00**
  - **what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?**

Example continued

- Represent the situation by creating a separate thread for each person to do the withdrawals
  - have both threads run on the same bank mainframe:

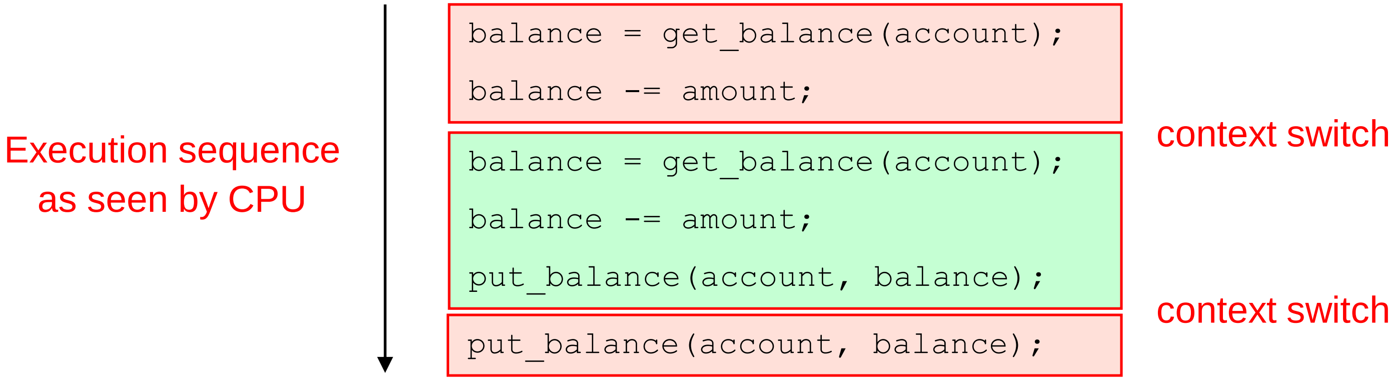
```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

- What’s the problem with this?
  - what are the possible balance values after this runs?

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:



```
balance = get_balance(account);
balance -= amount;
```

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

- What's the account balance after this sequence?
  - who's happy, the bank or you? ;)

The crux of the matter

- **The problem is that two concurrent threads (or processes) access a shared resource (account) without any synchronization**
  - **creates a race condition**
    - *output is non-deterministic, depends on timing*
- **We need mechanisms for controlling access to shared resources in the face of concurrency**
  - **so we can reason about the operation of programs**
    - essentially, re-introducing determinism
- **Synchronization is necessary for any shared data structure**
  - **buffers, queues, lists, hash tables, …**

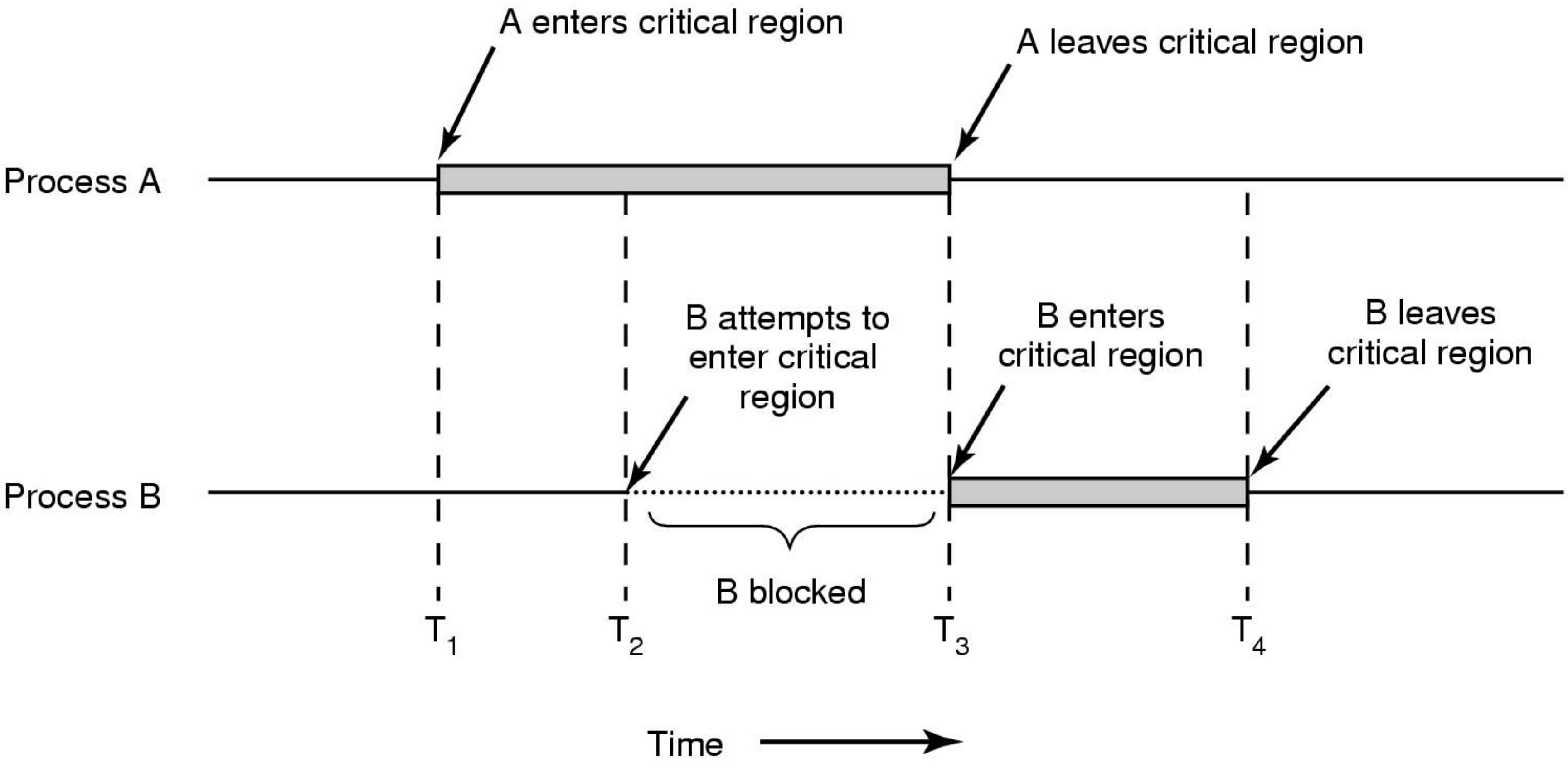
When are Resources Shared?

- **Local variables are not shared**
  - **refer to data on the stack, each thread has its own stack**
  - **never pass/share/store a pointer to a local variable on another thread’s stack**
- **Global variables are shared**
  - **stored in the static data segment, accessible by any thread**
- **Dynamic objects are shared**
  - **stored in the heap, shared if you can name it**
    - in C, can conjure up the pointer
      - e.g. void *x = (void *) 0xDEADBEEF
    - in Java, strong typing prevents this
      - must pass references explicitly

2.3.2 Critical Regions

- We want to use mutual exclusion (mutex) to synchronize access to shared resources
  - Way of making sure if one thread is using shared resource, other processes are excluded from doing the same
- Critical Region (Section)
  - That part of code where shared resources are accessed
- Use mutex to synchronize execution of critical section
  - only one thread at a time should execute in the critical section
  - all other threads should be forced to wait on entry
  - when a thread leaves a critical section, another can enter

Critical Regions



Mutual exclusion using critical regions

Critical Section Problem (CSP) Requirements

- **Critical section problem have to solve the following requirements**
  - **mutual exclusion**
    - at most one thread is in the critical section
  - **bounded waiting (no starvation)**
    - if thread T is waiting on the critical section, then T will eventually enter the critical section
      - assumes threads eventually leave critical sections
  - **progress**
    - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
  - **No assumption made regarding speed and/or number of CPUs**
  - **performance**
    - the overhead of entering and exiting the critical section is small with respect to the work being done within it

Mechanisms for Solving the Critical Section Problem

- **Locks**
  - very primitive, minimal semantics; used to build others
- **Semaphores**
  - basic, easy to get the hang of, hard to program with
- **Monitors**
  - high level, requires language support, implicit operations
  - easy to program with; Java "`synchronized()`" as example
- **Messages**
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems

2.3.3 Mutual Exclusion with Busy Waiting

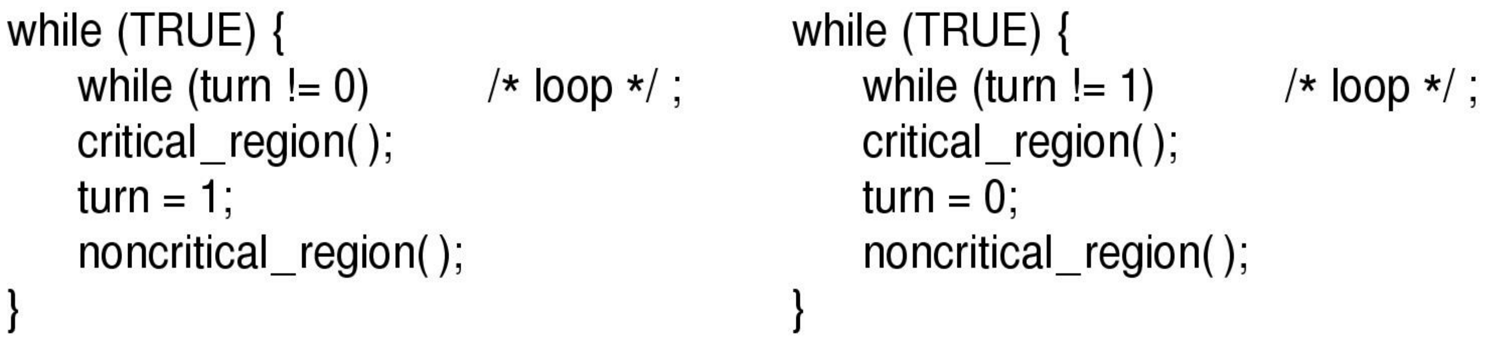
- Disabling Interrupts
  - Disable all interrupts just after entering its critical region and re-enable them just before leaving it
  - Unwise to give user processes this power
  - Not effective for multiprocessors

- Lock Variables
  - Software solution : use of shared lock variable
  - 0 means OK to enter
  - 1 means another thread is in CS
  - Any problems?

```
While (lock == 1);
lock = 1;
Critical_region()
lock = 0;
```

Strict Alteration

- Busy waiting
  - Continuously testing a variable until some value appears
  - Wastes CPU time - Used when a short waiting time expected
- Spin lock: a lock that uses busy waiting
- Violates condition 3 - progress condition
- A process is blocked by the other not in its critical region
- Requires that two processes strictly alternate in entering their critical regions



```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

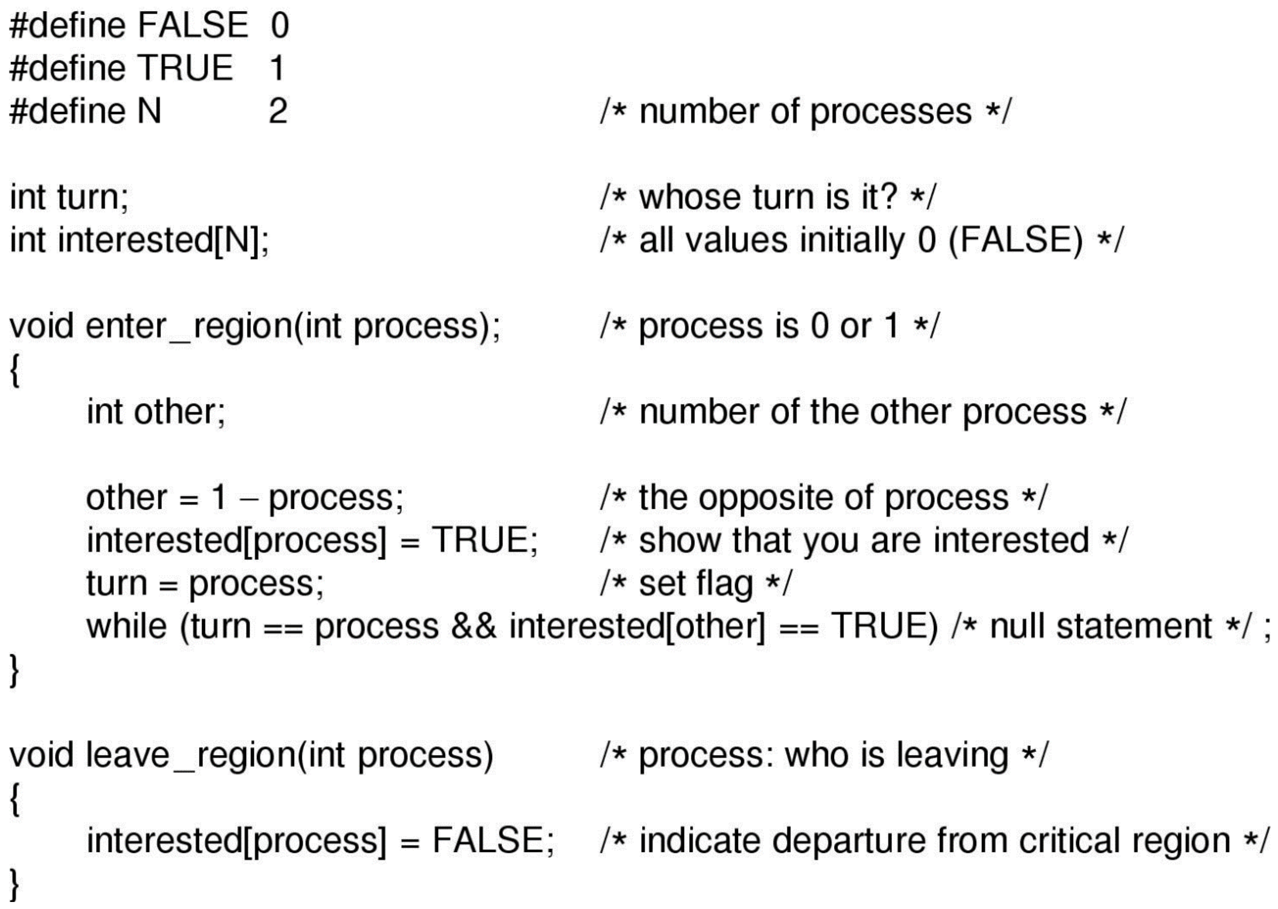
(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

◆Peterson’s Solution:

- **Other Software solution : Dekker’s algorithm etc.**
- **Hard to understand?, error?, performance etc.**



```
#define FALSE  0
#define TRUE   1
#define N      2                     /* number of processes */

int turn;                            /* whose turn is it? */
int interested[N];                   /* all values initially 0 (FALSE) */

void enter_region(int process);      /* process is 0 or 1 */
{
    int other;                       /* number of the other process */

    other = 1 – process;             /* the opposite of process */
    interested[process] = TRUE;      /* show that you are interested */
    turn = process;                  /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)       /* process: who is leaving */
{
    interested[process] = FALSE;     /* indicate departure from critical region */
}
```

◆TSL (Test and Set Lock) instruction

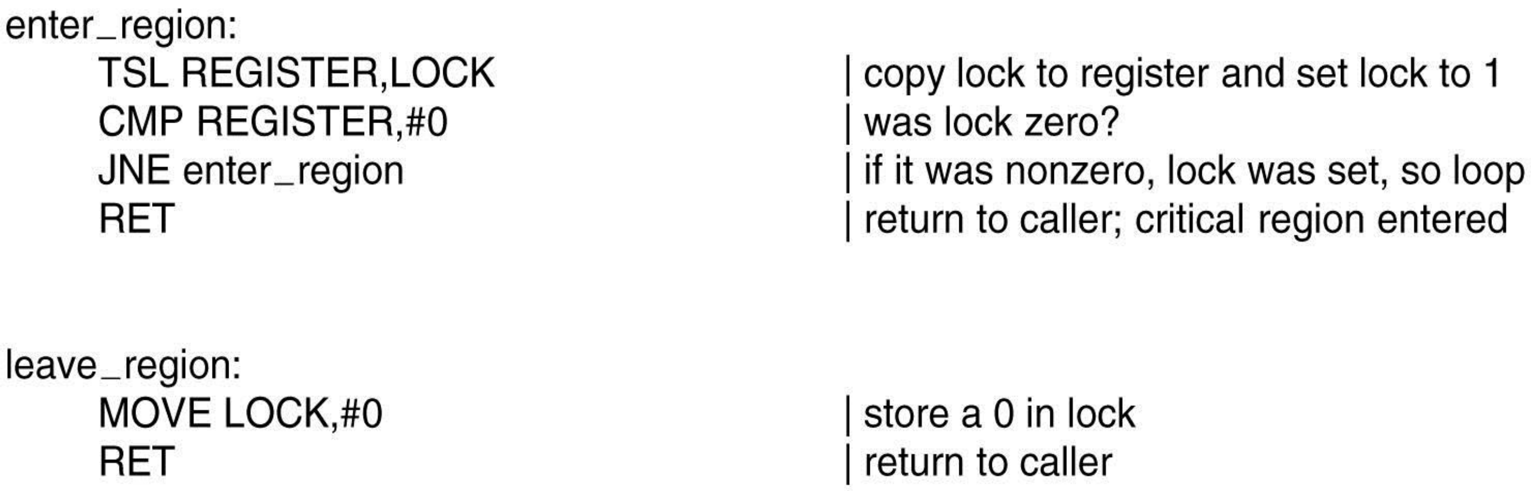
- **Hardware support**
  - Test-and-Set, Swap, Compare-and-swap atomic instructions
- **What is an atomic instruction?**
  - An instruction that finishes what it started
  - Indivisible operation
  - In most systems
    - Reference of word units and assignment (load/store) operations are atomic
    - Double precision floating-point or string operations are examples of non-atomic operations
- **Test-and Set instruction**

```
bool test_and_set(bool *flag){
  bool old = *flag;
  *flag = True;
  return old;
}
```

- **Solution**
  - *lock* : shared variable

```
lock = false
repeat
         while Test-and-Set(lock) do no-op;
         Critical Section
         lock = false;
         Remainder Section
until false
```

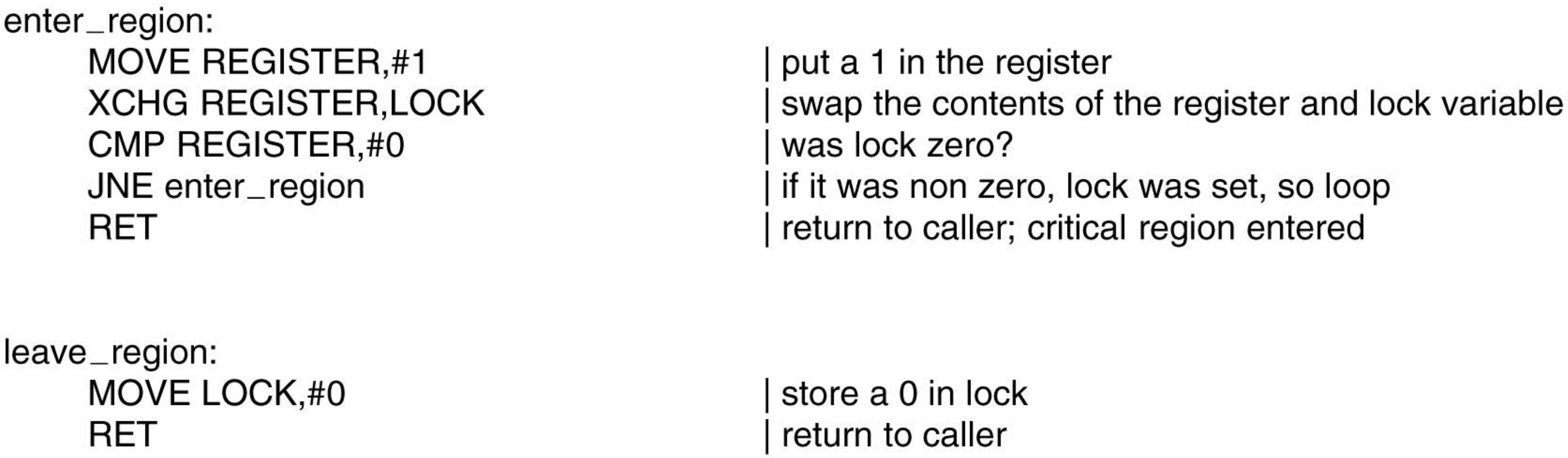
The TSL Instruction (1)



```
enter_region:
        TSL REGISTER,LOCK                  | copy lock to register and set lock to 1
        CMP REGISTER,#0                    | was lock zero?
        JNE enter_region                   | if it was nonzero, lock was set, so loop
        RET                                | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                       | store a 0 in lock
        RET                                | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

The TSL Instruction (2)



```
enter_region:
        MOVE REGISTER,#1                  | put a 1 in the register
        XCHG REGISTER,LOCK                | swap the contents of the register and lock variable
        CMP REGISTER,#0                   | was lock zero?
        JNE enter_region                  | if it was non zero, lock was set, so loop
        RET                               | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                      | store a 0 in lock
        RET                               | return to caller
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

Problems with spinlocks

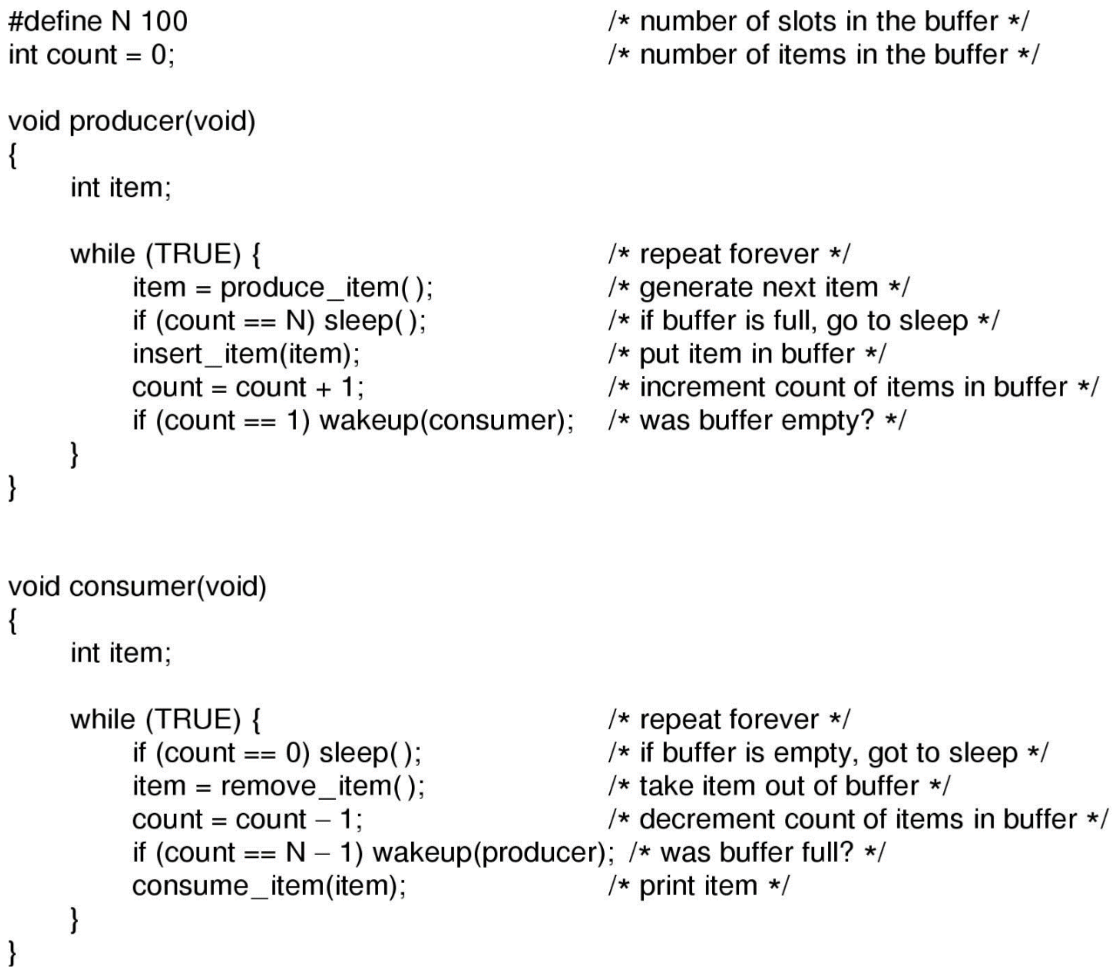
- **Horribly wasteful!**
  - **if a thread is spinning on a lock, the thread holding the lock cannot make process – *priority inversion problem***
- **How did lock holder yield the CPU in the first place?**
  - **calls yield( ) or sleep( )**
  - **involuntary context switch**
  - **Priority inversion problem**
    - Ex) a high priority process does busy waiting and a low priority process can not have the CPU to release a lock
- **Only want spinlocks as primitives to build higher-level synchronization constructs**

2.3.4 Sleep and Wakeup

- **Can we do it better than spinlocks?**
- **sleep and wakeup**
  - **sleep() : a *system call* that blocks the caller until a wakeup() call is made by another process.**
  - **wakeup() : wake a blocked process up.**
  - ***No busy waiting ; OS supports.***
- **Producer-consumer problem**
  - **Also known as the bounded-buffer problem**
  - **Two processes share a common, fixed-size buffer**
    - producer
      - Puts information into the buffer
    - consumer
      - Takes the information out of the buffer

Sleep and Wakeup

- **A failed Producer-consumer problem solution (Fig 2-27)**
  - **Race condition occurs when a wakeup set to a process that is not(yet) sleeping is lost. (*checking the count value and sleep/wakeup are separable ops!*)**



```
#define N 100                             /* number of slots in the buffer */
int count = 0;                            /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                       /* repeat forever */
          item = produce_item( );         /* generate next item */
          if (count == N) sleep( );       /* if buffer is full, go to sleep */
          insert_item(item);              /* put item in buffer */
          count = count + 1;              /* increment count of items in buffer */
          if (count == 1) wakeup(consumer); /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                       /* repeat forever */
          if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
          item = remove_item( );          /* take item out of buffer */
          count = count − 1;              /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);             /* print item */
     }
}
```

2.3.5 Semaphores

- Semaphore : a synchronization primitive
  - higher level than locks
  - invented by Dijkstra in 1968, as part of the THE os
- A semaphore is: *non-negative* integers
  - A variable that is manipulated atomically through two operations, down and up (or P and V)
  - down(semaphore): decrement, block until semaphore <= 0
    - also called P(), after Dutch word for test, also called wait()
    - If *sem* > 0, then decrement *sem* by 1 , otherwise "wait"
    - If awakened, then complete the down() ops
  - up(semaphore): increment, allow another to enter
    - also called V(), after Dutch word for increment, also called signal()
    - Inecrement *sem* by 1 and wake up a thread waiting in down()
    - Select one thread if multiple threads waiting in down()
  - Each operation above is indivisible atomic action.
    - Normally implemented as system calls where all interrupts are briefly disabled
    - Atomic actions : a group of actions either all performed without interruption or not performed at all

Two types of semaphores

- **Binary semaphore (aka mutex semaphore)**
  - guarantees mutually exclusive access to resource
  - only one thread/process allowed entry at a time
  - **counter is initialized to 1 (semaphore value changes between 0 and 1)**
- **Counting semaphore (aka counting semaphore)**
  - represents a resources with many units available
  - allows threads/process to enter as long as more units are available
  - **counter is initialized to N**
    - N = number of units available

Semaphores

Solving the Producer-Consumer Problem using Semaphores

- **For mutual exclusion**
  - Guarantees that only one process at a time reads or writes the buffer and the associated variable
  - ***mutex***
    - Makes sure the producer and consumer do not access the buffer at the same time (*the buffer is a shared object!!*)
    - binary semaphore
- **For synchronization**
  - Guarantees that the producer stops running when the buffer is full, and the consumer stops running when it is empty
  - ***full***
    - Counts the number of slots that are full
  - ***empty***
    - Counts the number of slots that are empty
    - Both are counting semaphores

Bounded Buffer using Semaphores

```
var mutex: semaphore = 1      ;mutual exclusion to shared data
    empty: semaphore = n      ;count of empty buffers (all empty to start)
    full: semaphore = 0       ;count of full buffers (none full to start)

producer:
          <produce the item>
    down(empty)     ; one fewer buffer, block if none available
    down(mutex)     ; get access to pointers
          <add item to buffer>
    up(mutex) ; done with pointers
    up(full) ; note one more full buffer

consumer:
    down(full) ;wait until there's a full buffer
    down(mutex) ;get access to pointers
          <remove item from buffer>
    up(mutex) ; done with pointers
    up(empty) ; note there's an empty buffer
          <use the item>
```

Problems with Semaphores

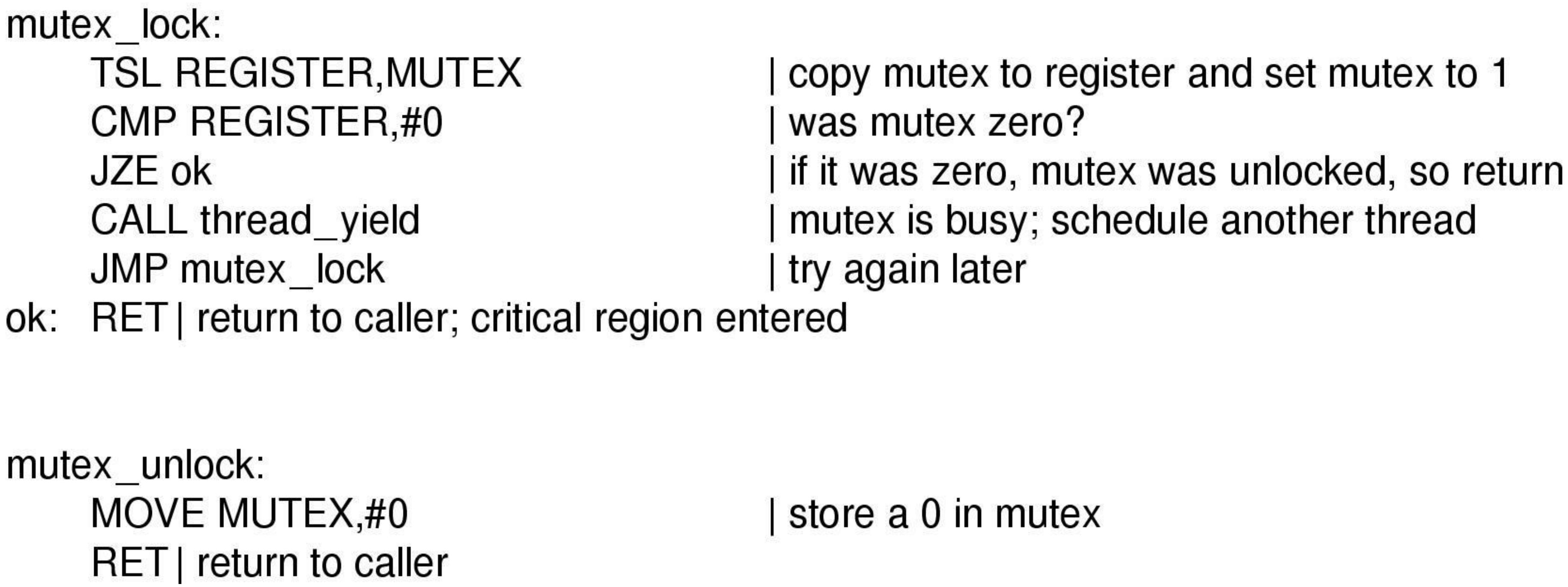
- **They can be used to solve any of the traditional synchronization problems, but:**
  - semaphores are essentially shared global variables
    - can be accessed from anywhere (bad software engineering)
  - there is no connection between the semaphore and the data being controlled by it
  - used for both critical sections (mutual exclusion) and for coordination (synchronization)
  - **no control over their use, no guarantee of proper usage**
    - Ex) down(mutex) followed by down(empty) !
- **Thus, they are prone to bugs**
  - **another (better?) approach: use programming language support like Monitors**

2.3.6 Mutexes

- **Simplified version of the semaphore**
- **A variable that can be in one of two states:**
  - **Unlocked : 0**
  - **Locked : non-zero**
- **Usage**

```
mutex_lock
    critical_region
mutex_unlock
```

Mutexes



```
mutex_lock:
        TSL REGISTER,MUTEX                      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0                         | was mutex zero?
        JZE ok                                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield                       | mutex is busy; schedule another thread
        JMP mutex_lock                          | try again later
ok:     RET | return to caller; critical region entered


mutex_unlock:
        MOVE MUTEX,#0                           | store a 0 in mutex
        RET | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*

(compare it with the previous example on TSL)

Busy waiting ? No? (thread_yield!)

Futexes (Linux)

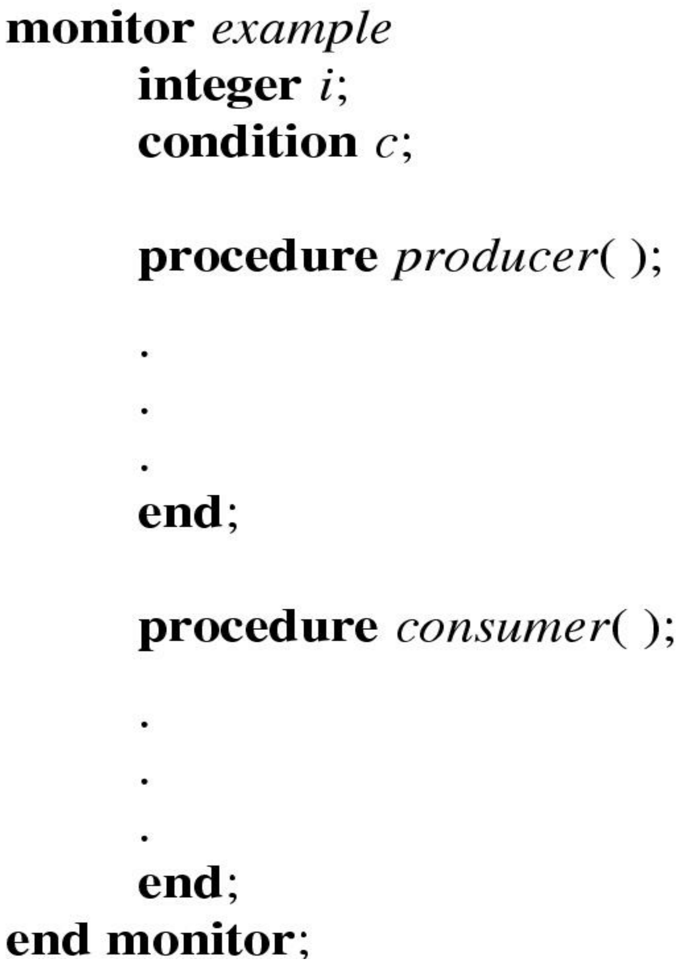
- Consists of a Kernel service and a user library
- A thread grabs the locks by an *atomic* "*decrement and test*"
  - *If succeeds, no kernel action is involved* *(completely done in user space)*
- If fails, use a *system call* to put the thread on the wait queue in the kernel
- Unlocking thread release the lock with *an atomic "increment and test"*
  - *If other threads are blocked on the lock, let the kernel know that it may unblock threads blocked on the kernel wait queue*

- **Sharing data between processes for Semaphores etc… (*compared to threads*)**
  - **The shared data structures, such as the semaphores, can be stored in the kernel and only accessed via system calls.**
  - **Most modern operating systems(including UNIX and Windows) offer a way for processes to share some portion of their address space with other processes.**
  - **If nothing else is possible, a shared file can be used.**

2.3.7 Monitors

- A programming language construct that supports controlled access to shared data
  - synchronization code added by compiler, enforced at runtime
  - why does this help?
- Monitor is a software module that encapsulates:
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent processes that invoke those procedures
- Monitor protects the data from unstructured access
  - guarantees only access data through procedures in monitor, hence in legitimate ways

Monitors



```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Example of a monitor

Monitor facilities

- **Mutual exclusion**
  - **only one process can be executing inside at any time**
    - thus, synchronization implicitly associated with monitor
  - **if a second process tries to enter a monitor procedure, it blocks until the first has left the monitor**
    - more restrictive than semaphores!
    - but easier to use most of the time

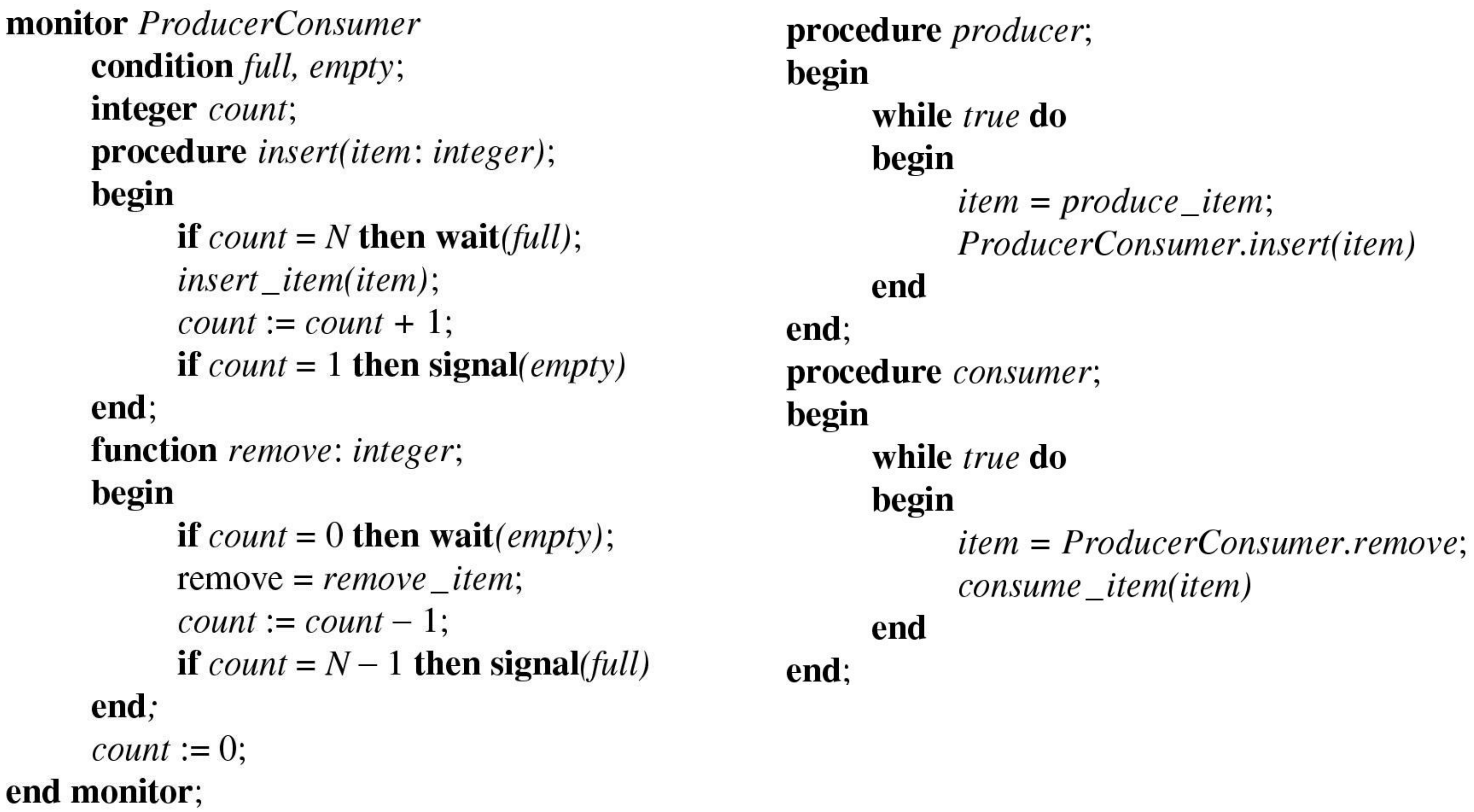
- **What happens when inside and finds itself not able to progress?**
  - **Need to wait for another process to provide result?**
  - **Need to wait for particular task to finish?**

Condition Variables

- **Condition variables are provided within monitor**
  - **condition variable can only be accessed from inside monitor**
- **A place to wait if not able to progress; sometimes called a rendezvous point**

- **Three operations on condition variables**
  - **wait(c)**
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have wait queues
  - **signal(c)**
    - wake up at most one waiting process/thread
    - if no waiting processes, signal is lost - different than semaphores: no history!, not counter!
    - *Which one will continue first after signal? Signaler or Waiter?*
  - **broadcast(c)**
    - wake up all waiting processes/threads

Producer/Consumer with Monitors



```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count − 1;
                if count = N − 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

- only one monitor procedure active at one time
- buffer has N slots
- *Similar to sleep/wakeup(), but no race conditions! Why?*

Three Kinds of Monitors

- **Hoare monitors: signal(c) means**
  - **waiter runs immediately and signalers blocks**
  - **condition guaranteed to hold when waiter runs**
    - but, signaler must restore monitor invariants before signaling!
  - *monitor invariants : assertion that must hold when no thread in monitor to avoid race conditions*
- **Hansen monitors : signal(c) means**
  - **signalers must exit the monitor immediately and waiter runs**
  - **Conceptually simpler and easier to implement**
- **Mesa monitors: signal(c) means**
  - **waiter is made ready, but the signaler continues**
    - waiter runs when signaler leaves monitor (or waits)
    - condition is not necessarily true when waiter runs again
  - **signaler need not restore invariant until it leaves the monitor**
  - **being woken up is only a hint that something has changed**
    - must recheck conditional case

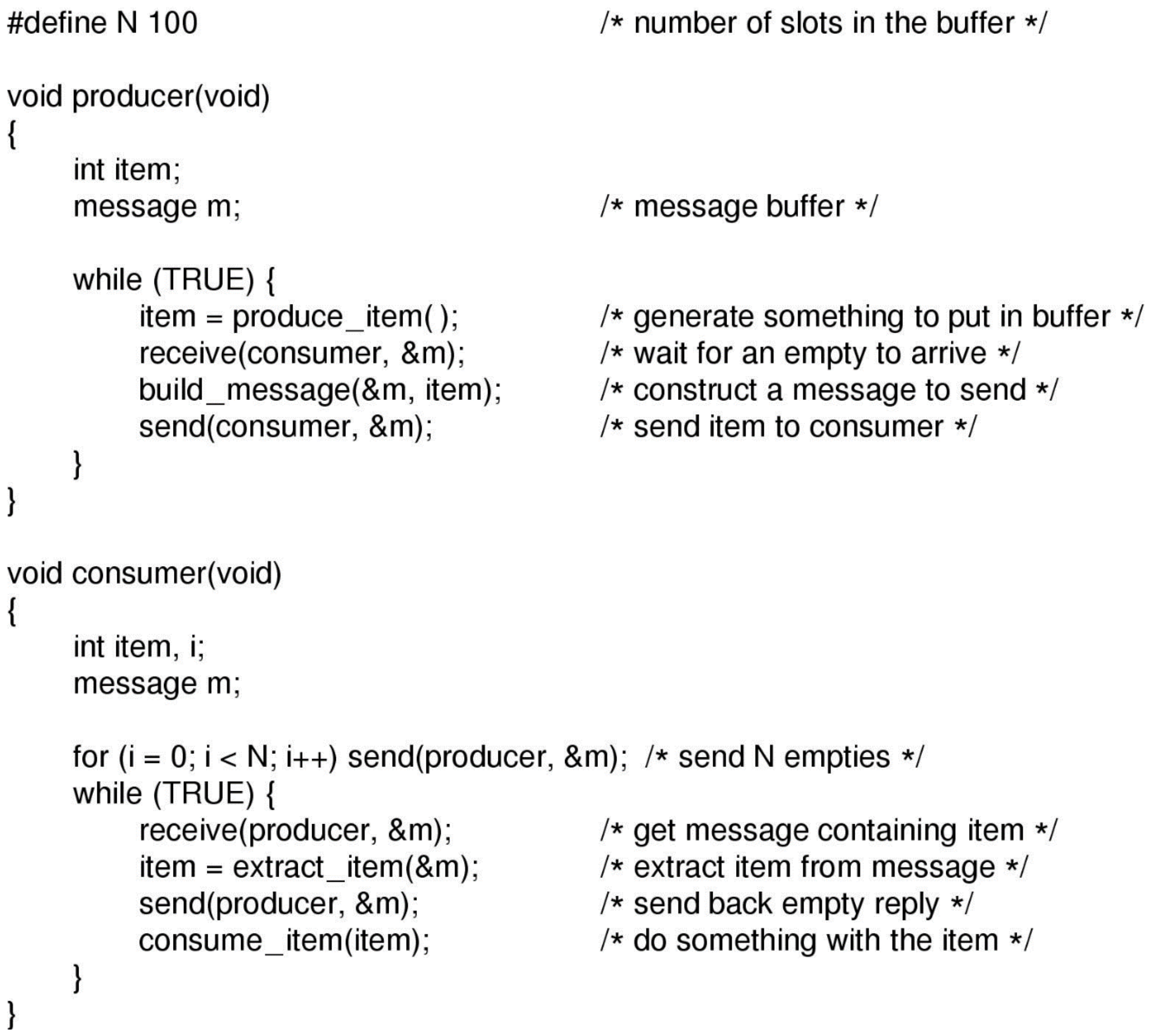
Problems with Monitors

- Monitors are a programming language concepts
  - Need a language that has them built in.
  - Should be supported by compilers
  - Thus, can not use monitors with C or C++ etc.
- Inapplicable to distributed environment
  - multiple CPUs, each with its own private memory, connected by a local network
  - The same problem with semaphores too.

2.3.8 Message Passing

- **Method of interprocess communication**
  - **Uses send and receive system calls**
  - **send(destination, &message)**
    - Sends a message to a given destination
  - **receive(source, &message)**
    - Receives a message from a given source
    - If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.
- **Design issues**
  - **Messages can be lost by the network.**
    - Acknowledgement, sequence number
  - **How processes are named**
  - **Authentication**
  - **Performance issue when the sender and receiver are on the same machine**

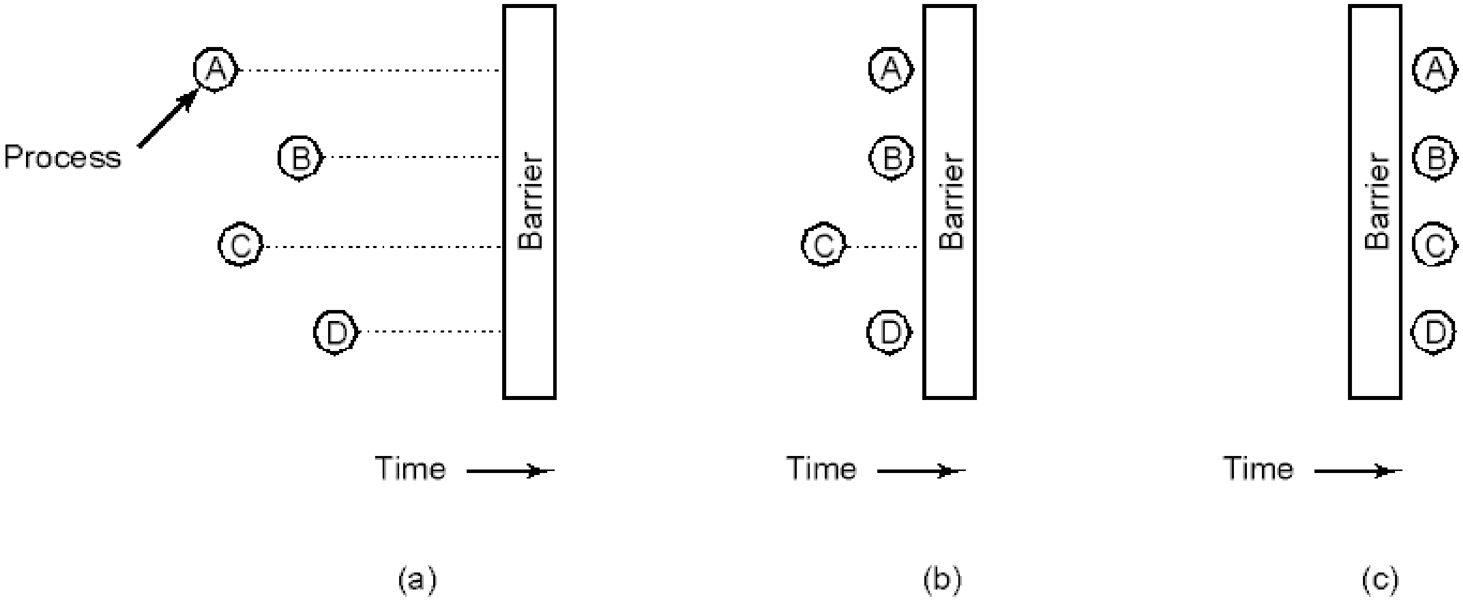
Message Passing



```
#define N 100                              /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                             /* message buffer */

    while (TRUE) {
        item = produce_item( );            /* generate something to put in buffer */
        receive(consumer, &m);             /* wait for an empty to arrive */
        build_message(&m, item);           /* construct a message to send */
        send(consumer, &m);                /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);             /* get message containing item */
        item = extract_item(&m);           /* extract item from message */
        send(producer, &m);                /* send back empty reply */
        consume_item(item);                /* do something with the item */
    }
}
```

The producer-consumer problem with N messages

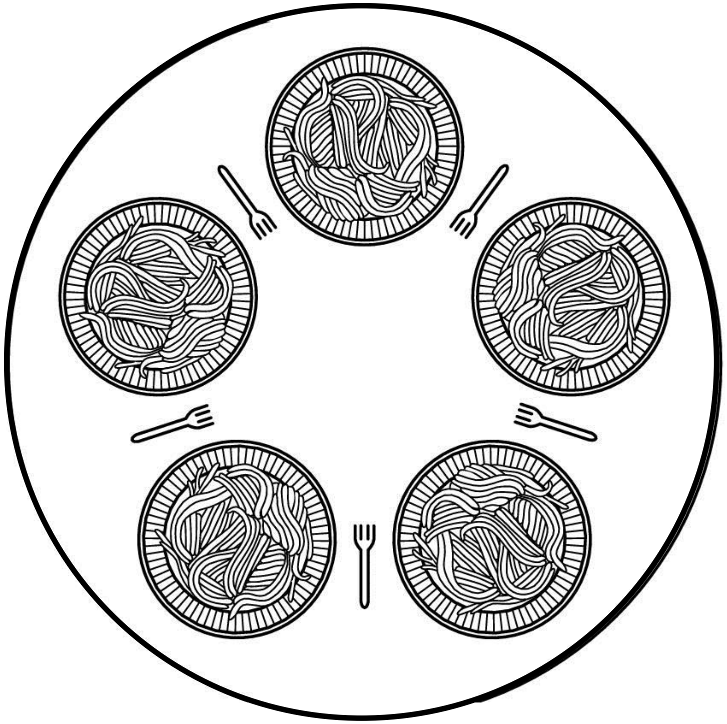
2.3.9 Barriers



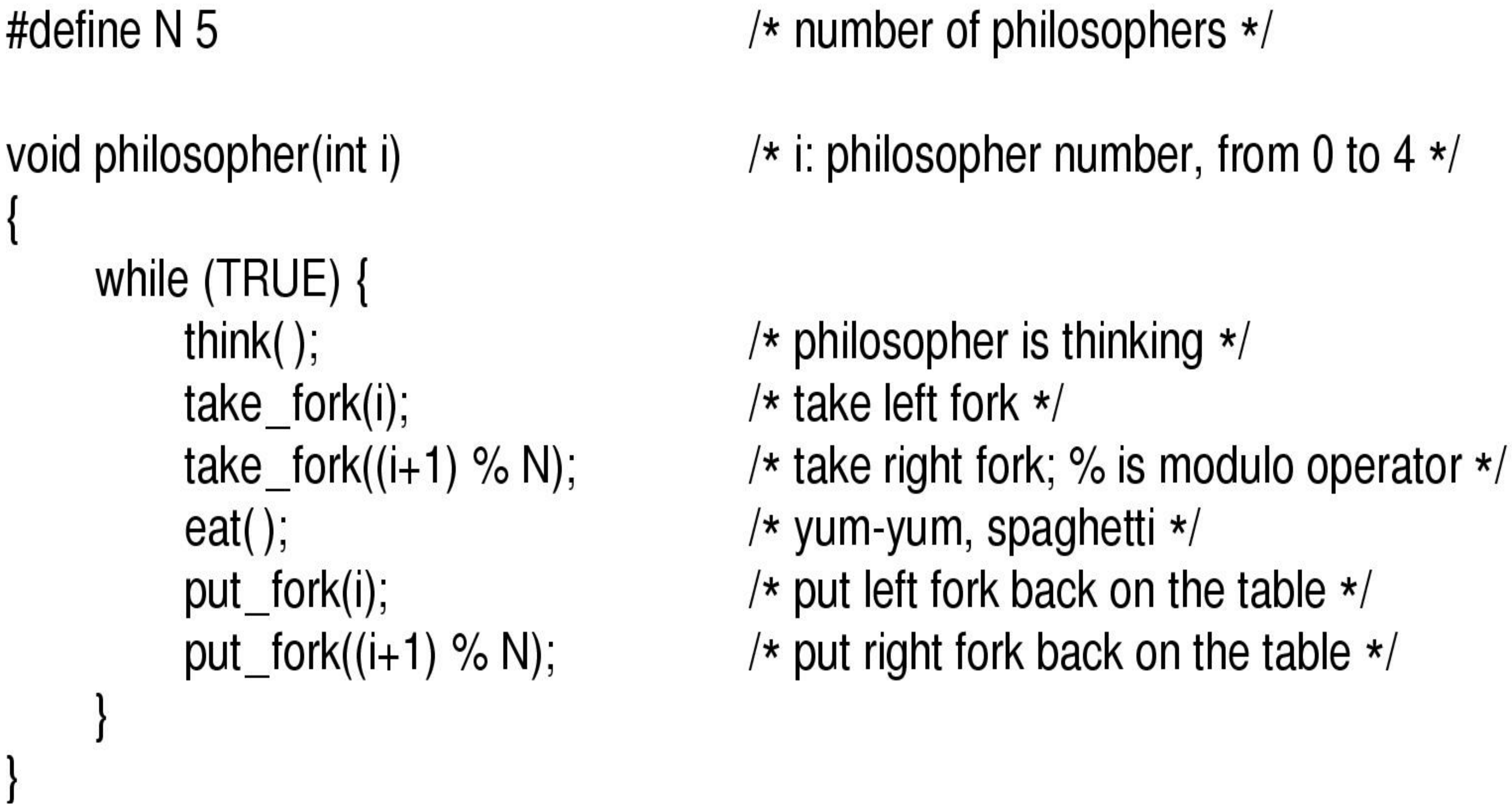
- **Use of a barrier**
  - **processes approaching a barrier**
  - **all processes but one blocked at barrier**
  - **last process arrives, all are let through**

2.5 Classical IPC Problems:
2.5.1 The Dining Philosophers Problem

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



NonSolution to the Dining Philosophers Problem



```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                    /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat( );                      /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }
}
```

2.5.2 The Readers/Writers Problem

- Basic problem:
  - object is shared among several processes
  - some read from it
  - others write to it
- We can allow multiple readers at a time
  - why?
- We can only allow one writer at a time
  - why?

2.5.3 The Sleeping Barber Problem

2.4 Scheduling

- **In discussing process management, we talked about context switching between threads/process on the ready queue**
  - **but, we glossed over the details of which process or thread is chosen next**
  - **making this decision is called scheduling**
    - scheduling is policy
    - context switching is mechanism
- **We will look at:**
  - **the goals of scheduling**
    - starvation etc…
  - **well-known scheduling algorithms**
    - standard UNIX scheduling

2.4.1 Introduction to Scheduling

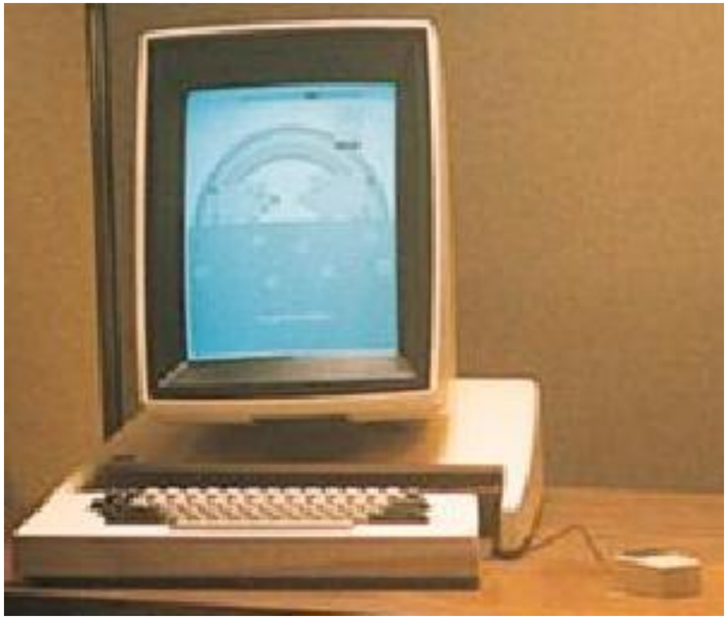
- Batch processing
  - So simple
- Time sharing/Multiprogramming
  - Scheduling become complex and important
- Combination of the two
- Key: CPU is a scarce resource

1975: Honeywell 6180

1968: Honeywell 645

With Advent of PCs

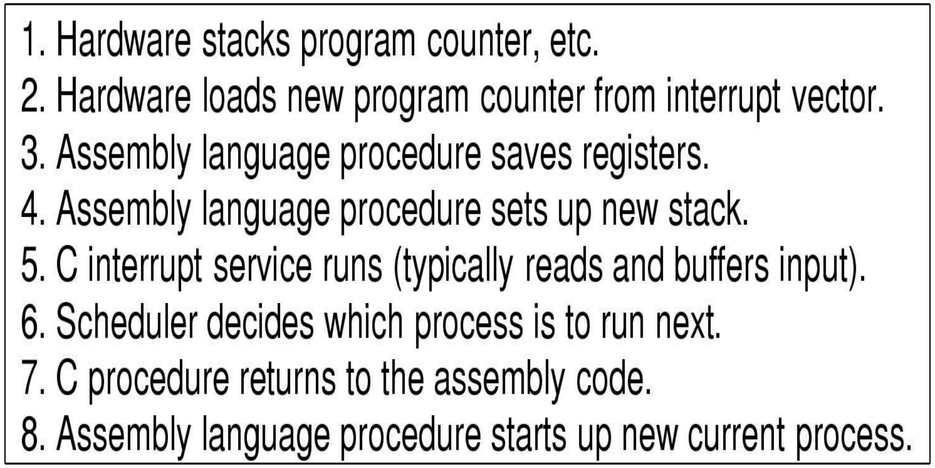
- Only one job is active (usually)
- CPU is no longer scarce resource



< $50K

What about servers?

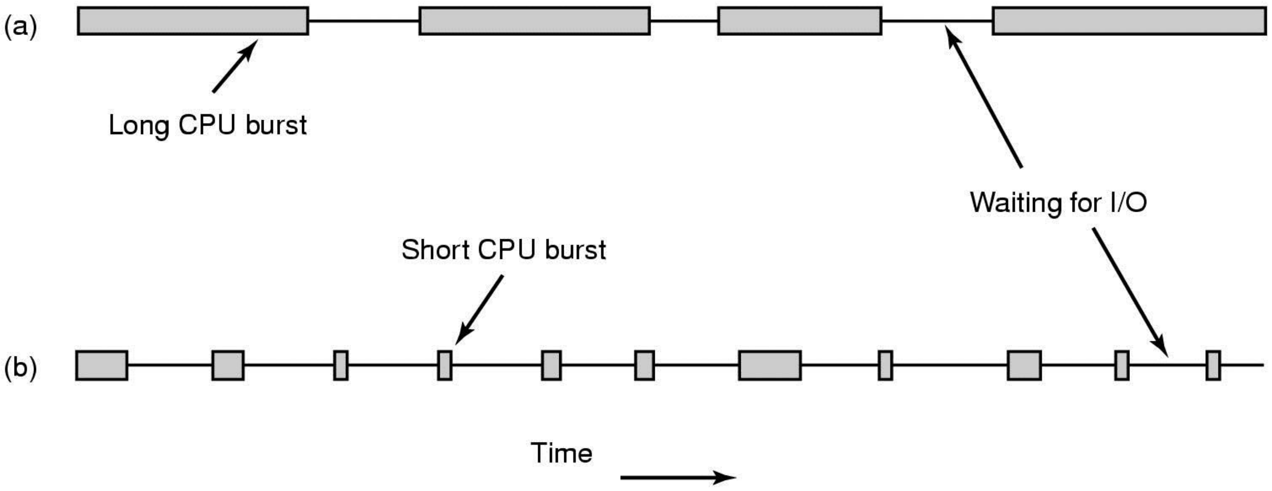
- Does scheduling for CPU matter? YES again
- User perception is important
  - Email vs screen update
- Scheduler need to consider the cost of context switches
- What does a context switch involve
  - Switch from user-mode to kernel-mode
  - Save current state
  - Save memory map
  - Select next process
  - Reload MMU
  - → Invalidate entire memory cache
  - → Chews up CPU time



1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Process Behavior

- **Compute-bound - (a)**
- **I/O-bound - (b)**
  - **I/O: A process enters the blocked state waiting for an external device to complete its work**



- **What happens as CPUs get faster?**

When to Schedule

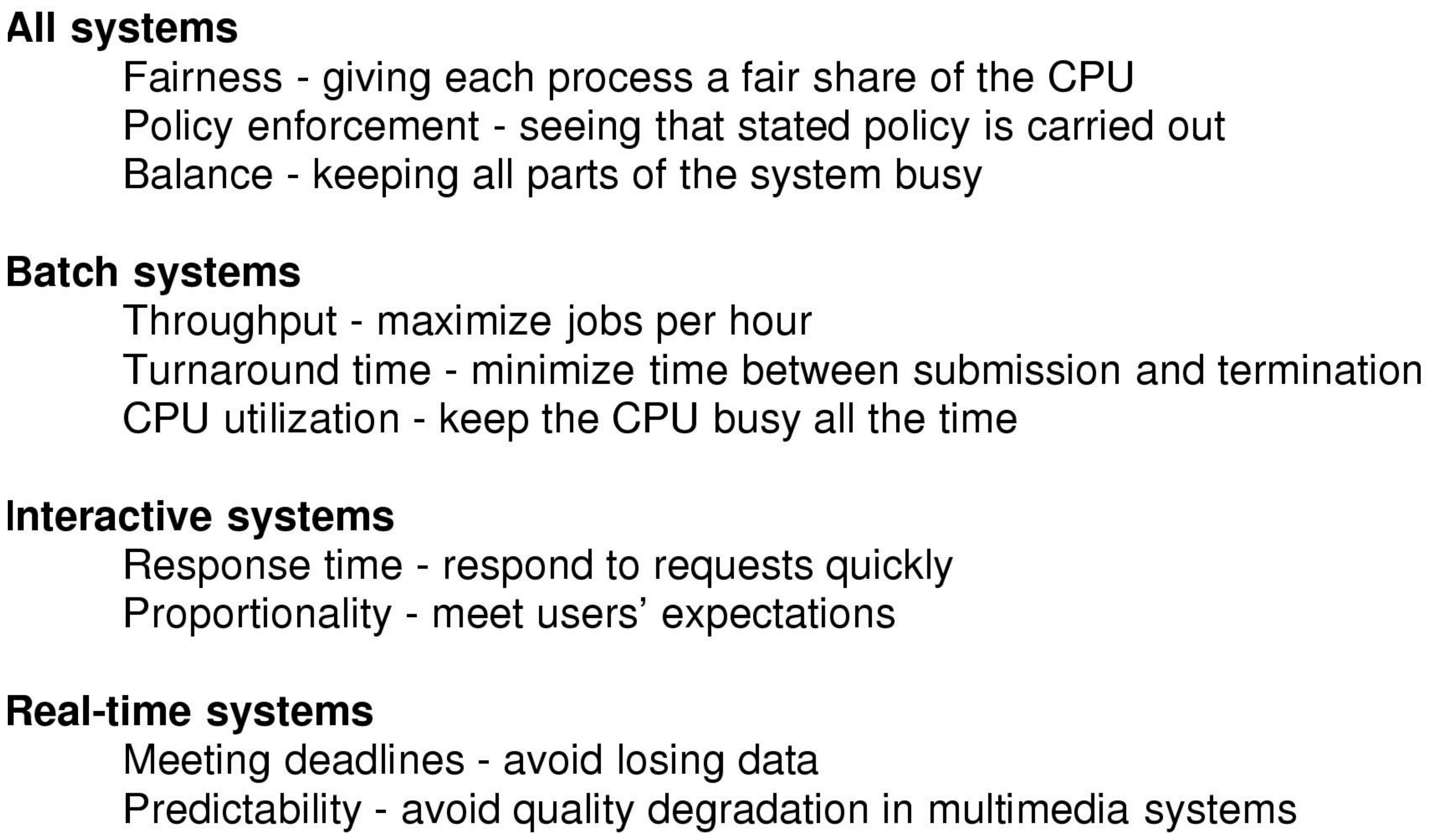
- **When a new process is created**
- **When a process exits**
- **When a process blocks**
- **When an I/O interrupt occurs**
- **When a clock interrupts**

- **Two categories of scheduling** wrt clock interrupts
  - **Nonpreemptive**
    - Picks a process to run and lets it run until it blocks or until it voluntarily releases the CPU
  - **Preemptive**
    - If a process is running at the end of the time interval, it is suspended and the scheduler picks another process to run

Three Environments for Scheduling

- Batch
  - Nonpreemptive algorithms or preemptive algorithms with long time periods for each process
  - Reduces process switches and improves performance
- Interactive
  - Preemptive algorithms is essential to keep one process from hogging the CPU and denying service to the others
- Real time
  - Preemption is sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly

Scheduling Algorithm Goals



All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm Goals

*Multiprogramming and Scheduling

- Multiprogramming increases resource utilization and job throughput by overlapping I/O and CPU
  - today: look at scheduling policies
    - which process/thread to run, and for how long
  - schedulable entities are usually called jobs
    - processes, threads, people, disk arm movements, …
- **There are two time scales of scheduling the CPU:**
  - **long term: determining the multiprogramming level**
    - how many jobs are loaded into primary memory
    - act of loading in a new job (or loading one out) is swapping
  - **short-term: which job to run next to result in “good service”**
    - happens frequently, want to minimize context-switch overhead
    - good service could mean many things

Scheduling

- **The scheduler is the module that moves jobs from queue to queue**
  - **the scheduling algorithm determines which job(s) are chosen to run next, and which queues they should wait on**
  - **the scheduler is typically run when:**
    - a job switches from running to waiting
    - when an interrupt occurs
      - especially a timer interrupt
    - when a job is created or terminated
- **There are two major classes of scheduling systems**
  - **in preemptive systems, the scheduler can interrupt a job and force a context switch**
  - **in non-preemptive systems, the scheduler waits for the running job to explicitly (voluntarily) block**

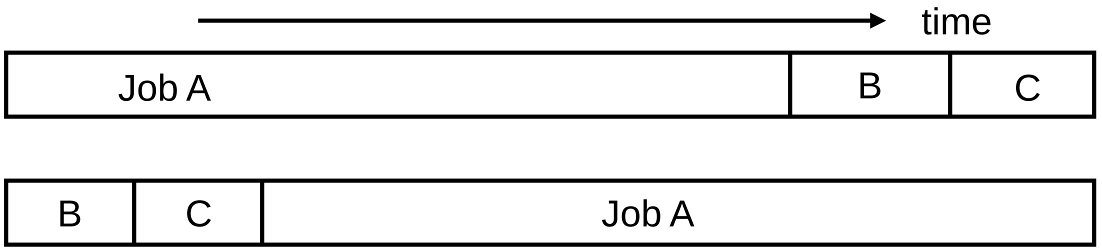
Scheduler Non-goals

- Schedulers typically try to prevent starvation
  - starvation occurs when a process is prevented from making progress, because another process has a resource it needs
- A poor scheduling policy can cause starvation
  - e.g., if a high-priority process always prevents a low-priority process from running on the CPU
- Synchronization can also cause starvation
  - we'll see this in a future class
  - roughly, if somebody else always gets a lock I need, I can't make progress

2.4.2 Scheduling in Batch Systems
Algorithm #1: FCFS/FIFO

- **First-come first-served (FCFS)**
  - **jobs are scheduled in the order that they arrive**
  - **“real-world” scheduling of people in lines**
    - e.g. supermarket, bank tellers, McDonalds, …
  - **typically non-preemptive**
    - no context switching at supermarket!
  - **jobs treated equally, no starvation**
    - except possibly for infinitely long jobs
- **Sounds perfect!**
  - **what’s the problem?**

FCFS picture

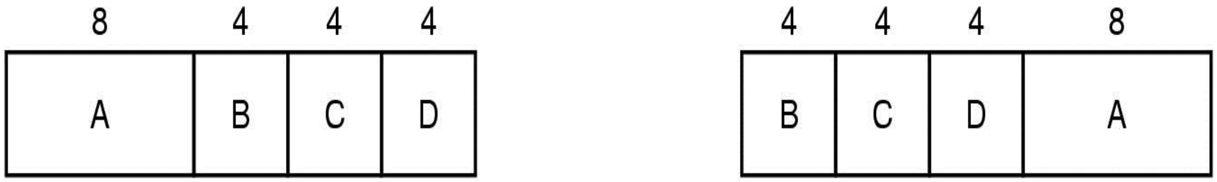


Problems:

- **average response time and turnaround time can be large**
  - e.g., small jobs waiting behind long ones
  - results in high turnaround time
- **may lead to poor overlap of I/O and CPU**
  - Ex) one compute-bound job runs 1 sec at a time and reads disk and Many I/O-bound jobs perform 1000 disk reads
    - FCFS(non-preemptive) : each I/O job take 1000 sec
    - Compute-bound job preemtped every 10ms : each I/O job finish in 10 sec

Algorithm #2: SJF

- Shortest job first (SJF)
  - choose the job with the smallest expected CPU burst
  - can prove that this has optimal min. average waiting time
- Non-preemptive
  - preemptive is called shortest remaining time next (SRTN)
- Sounds perfect!
  - what’s the problem here?



(a)

(b)

Avg. turnaround time :

$$\text{In (a), } (8+12+16+20)/4=14 \text{ minutes}$$

$$\text{In (b), } (4+8+12+20)/4=11 \text{ minutes}$$

SJF Problem

- **Problem: impossible to know size of future CPU burst**
  - **from your theory class, equivalent to the halting problem**
  - **can you make a reasonable guess?**
    - yes, for instance looking at past as predictor of future
    - but, might lead to starvation in some cases!
- **Another problem :**
  - **Optimal only when all jobs are *available simultaneously*.**
  - **Ex) Job A,B,C,D,E with run times of 2,4,1,1,1 and arrival times of 0,0, 3,3,3 : SJF results in the order A,B,C,D,E, but the order B,C,D,E,A has the best result.**

Algorithm #3: SRTN

- Shortest Remaining Time Next(SRTN)
  - A preemptive version of shortest job first
  - The run time has to be know in advance
  - New short jobs can get good service

2.4.3 Scheduling in Interactive Systems

Algorithm #4: Round Robin

- Round Robin scheduling (RR)
  - **ready queue is treated as a circular FIFO queue**
  - **each job is given a time slice, called a quantum**
    - job executes for duration of quantum, or until it blocks
    - time-division multiplexing (time-slicing)
  - **great for timesharing**
    - no starvation
    - can be preemptive or non-preemptive
- Sounds perfect!
  - **what’s wrong with this?**

RR problems

- **Problems:**
  - **what do you set the quantum to be?**
    - no setting is “correct”
      - if small, then context switch often, incurring high overhead
      - if large, then response time drops
      - Usually set a quantum to 20~50ms

Algorithm #5: Priority Scheduling

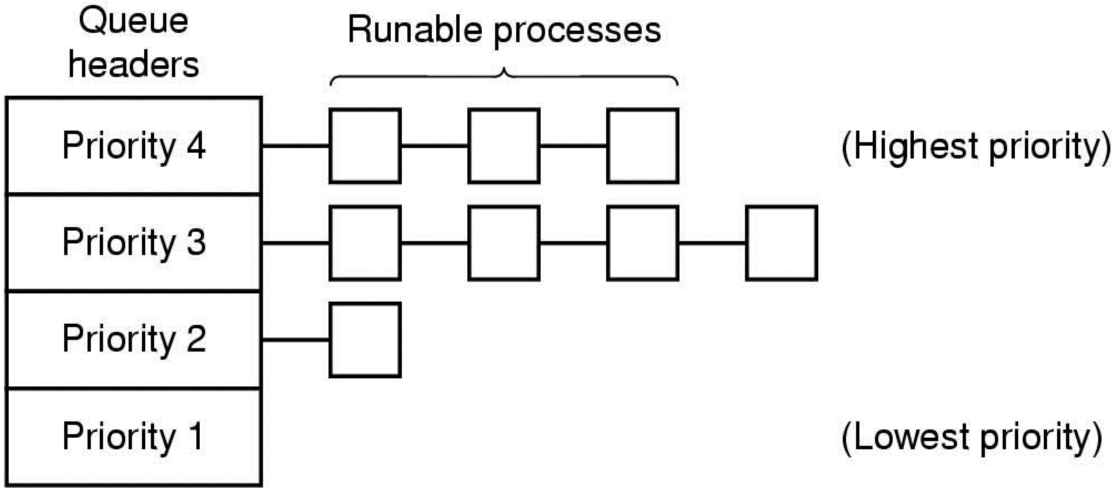
- **Assign priorities to jobs**
  - **choose job with highest priority to run next**
    - if tie, use another scheduling algorithm to break (e.g. FCFS)
  - **to implement SJF, priority = expected length of CPU burst**
- **Abstractly modeled as multiple “priority queues”**
  - **put ready job on queue associated with its priority**
- **Priority Assignment**
  - **Static**
    - Based on the user, price, etc. (cf. nice)
  - **Dynamic**
    - **Giving good service to I/O-bound processes**
    - Sets the priority to 1/f (f: fraction of the last quantum that a process used.)
- **Sound perfect!**
  - **what’s wrong with this?**

Priority Scheduling: problem

- **The problem: starvation**
  - **if there is an endless supply of high priority jobs, no low-priority job will ever run**
- **Solution: “age” processes over time**
  - **increase priority as a function of wait time**
  - **decrease priority as a function of CPU time**
    - decrease priority at each clock ticks when CPU is used or if the quantum is used up.
  - **many ugly heuristics have been explored in this space**

Priority Scheduling

- Example
  - Group processes into priority classes
  - Priority scheduling among the classes
  - Round-robin scheduling within each classes



Algorithm #6: Multiple Queues (and its variants)

- **Scheduling algorithms can be combined in practice**
  - **have multiple queues**
  - **pick a different algorithm for each queue**
  - **and maybe, move processes between queues**
- **Example: multi-level feedback queues (MLFQ)**
  - **multiple queues representing different job types**
    - batch, interactive, system, CPU-bound, etc.
    - lower priority queue has a longer quantum
  - **queues have priorities**
    - schedule jobs within a queue using RR
  - **jobs move between queues based on execution history**
    - “feedback”: switch from CPU-bound to IO-bound
      - A CPU-bound job moves to a lower priority queue when it use up its quantum
      - Interactive job moves to a upper priority queue.
- **Pop-quiz:**
  - **is MLFQ starvation-free?** No

Algorithm #8: Guaranteed Scheduling

- **Make real promises about performance and live up to them**
- **Guarantee 1/n of the CPU power**
  - **Keeps track of how much CPU each process has had since its creation**
  - **Computes**

$$\text{ratio} = \frac{\text{actual CPU time consumed}}{\text{CPU time entitled}}$$

    - E.g. 0.5 : a process has only had half of what it should have had
  - **Runs the process with the lowest ratio until its ratio has moved above its closest competitor**

Algorithm #9: Lottery Scheduling

- **Waldspurger and Weihl, 1994**
- **Another alg. that provides the performance assurance.**
- **Scheduler chose a lottery ticket at random and select the process holding the ticket for the resource**
- **Process holding a fraction *f* of the tickets will get about a fraction *f* of the resource in question**
  - **In contrast to the vague meaning of priority 40!**
- **Interesting properties**
  - **Highly responsive**
    - even a new process can get serviced right after it gets ready
  - **Processes may cooperate by exchanging tickets.**
  - **Proportionate scheduling is possible**
    - processes servicing different frame rates (10,20,and 25 frames/sec)
    - Then allocate 10,20, and 25 tickets respectively

Fair-share scheduling

- Share of resource based on owner, not process

2.4.4 Scheduling in Real-Time Systems

- Correctness + Timeliness
- Hard real-time
  - Absolute deadlines must be met
- Soft real-time
  - Missing an occasional deadline is undesirable, but nevertheless tolerable

- Process behavior is predictable and known in advance
- Real-time Events
  - Periodic : occurring at regular intervals
  - Aperiodic : occurring unpredictably

Scheduling in Real-Time Systems

- **Goal: schedulable?**
  - **Given**
    - *m* periodic events
    - event *i* occurs with period P_i and requires C_i seconds to handle
  - **Then the load can only be handled if the sum of the fraction of the CPU being used by each process <= 1.**

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

    - E.g. periods 100ms(CPU time 50ms), 200(30), 500(100)
      - To add a fourth event with a period of 1 sec,
        - CPU time <= 150ms
- **Static vs. dynamic real-time scheduling**
  - **Static : make scheduling decisions before the system starts running.**
  - **Dynamic : make scheduling decisions at run tim**