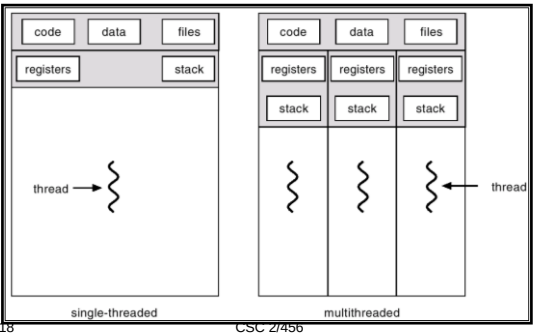
Last Class: Processes Vs. Threads

- Process
  - Single address space
  - Single thread of control for executing program
  - State information
    - Page tables, swap images, file descriptors, queued I/O requests, saved registers
- Threads
  - Separate notion of execution from the rest of the definition of a process
    - Program counter, stack of activation records, control block (e.g., saved registers/state info for thread management)
  - Other parts potentially shared with other threads

Processes and Threads

- Thread – a program in execution; without a dedicated address space.
- OS memory protection is only applied to processes.



Pthreads

- Each OS has its own thread package with different Application Programming Interfaces ⇒ poor portability.
- Pthreads
  - A POSIX standard API for thread management and synchronization.
  - API specifies behavior of the thread library, not the implementation.
  - Commonly supported in UNIX operating systems.

Thread Creation

```
int pthread_create
  (pthread_t *new_id,
  const pthread_attr_t *attr,
  void *(*func) (void *),
  void *arg)
```

- new_id: thread's unique identifier
- attr: ignore for now
- func: function to be run in parallel
- arg: arguments for function func

Example of Thread Creation

```
void *func(void *arg) {
   int        *I=arg;
   …..
}

void main()
{
   int X;    pthread_t      id;
   ….
   pthread_create(&id, NULL, func, &X);
   …
}
```


Pthread Termination

void pthread_exit(void *status)

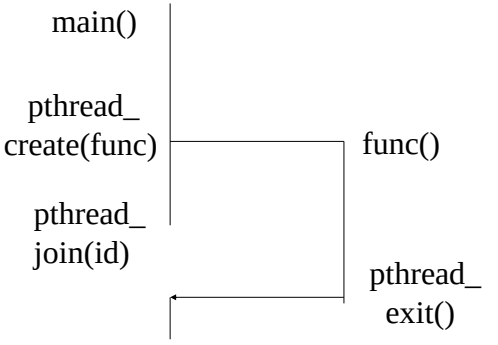
- Terminates the currently running thread.
- Is implicit when the function called in pthread_create returns.

Thread Joining

int pthread_join(
pthread_t new_id,
void **status)

- Waits for the thread with identifier new_id to terminate, either by returning or by calling pthread_exit().
- Status receives the return value or the value given as argument to pthread_exit().

Example of Thread Creation



Contention Scope

- Process contention scope – thread library schedules user threads onto light-weight processes (kernel-level thread)
  - Use priority as defined by user – no preemption of threads with same priority
- System contention scope – compete with all tasks and schedule kernel thread on a physical CPU
- pthreads: PTHREAD_SCOPE_PROCESS, PTHREAD_SCOPE_SYSTEM
  - pthread_attr_setscope
  - pthread_attr_getscope

Pthread Attributes

- Pthread_attr_init(pthread_attr_t *attr), destroy – initializes attr to default value
  - Scope – pthread_attr_setscope (&attr, SCOPE)
  - Stack size – pthread_attr_getstacksize, pthread_attr_setstacksize
  - Priority
  - Joinable or detached

Issues with the Threading Model

- Thread-local storage – what about globals?
- Stack management
- Interaction with fork and exec system calls
  - Two versions of fork?
- Signal handling – which thread should the signal be delivered to?
  - Synchronous
  - All
  - Assigned thread
  - Unix: could assign a specific thread to handle signals
  - Windows: asynchronous procedure calls, which are thread-specific

Multiprocessor Scheduling

- Disabling signals not sufficient
- Acquire scheduler lock when accessing any scheduler data structure, e.g.,

yield:

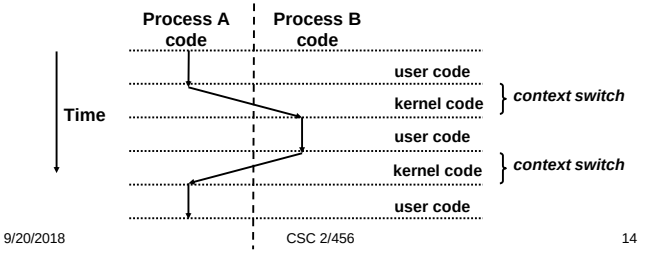
```
disable_signals
acquire(scheduler_lock) // spin lock
enqueue(ready_list, current)
reschedule
release(scheduler_lock)
re-enable_signals
```

CPU Scheduling

CS 256/456
Department of Computer Science
University of Rochester

Context Switching

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a *context switch.*



Thread Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

save all callee-saves registers on stack, including ra and fp

*current := sp

current := other

sp := *current

pop all callee-saves registers (including ra, but NOT sp!)

return (into different coroutine!)

Uniprocessor Scheduling

- Use Ready List to reschedule voluntarily (cooperative threading)

reschedule:
  - t : cb := dequeue(ready_list)
  - transfer(t)

yield:
  - enqueue(ready_list, current)
  - reschedule

sleep_on(q):
  enqueue(q, current)
  reschedule

Preemption

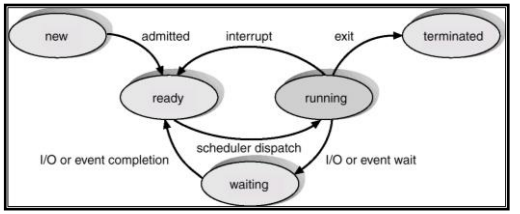
- Use timer interrupts or signals to trigger involuntary yields
- Protect scheduler data structures by disabling/reenabling prior to/after rescheduling

yield:

disable_signals
enqueue(ready_list, current)
reschedule
re-enable_signals

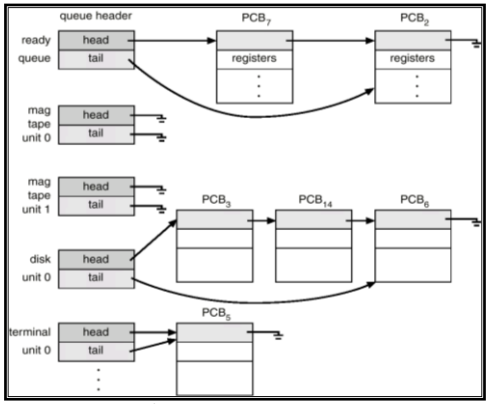
Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a process
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution



Queues for PCBs

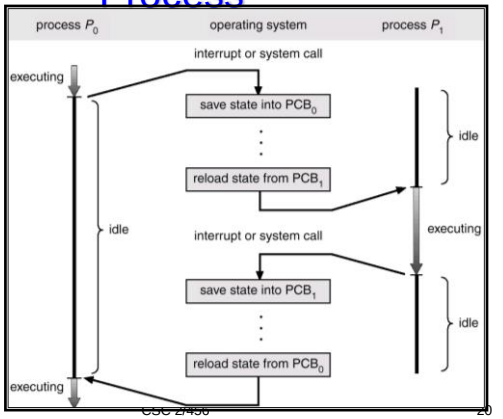
- Ready queue – set of all processes ready for execution.
- Device queues – set of processes waiting for an I/O device.
- Process migration between the various queues.



CPU Switch From Process to Process

When can the OS switch the CPU from one process to another?

Which one to switch to? - scheduling



CPU Scheduling

- Selects from among the processes/threads that are ready to execute, and allocates the CPU to it
- CPU scheduling may take place at:
  1. Hardware interrupt/software exception
  2. System calls
- *Nonpreemptive:*
  - Scheduling only when the current process terminates or not able to run further
- *Preemptive:*
  - Scheduling can occur at any opportunity possible

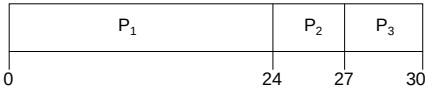
Scheduling Criteria

- Minimize turnaround time – amount of time to execute a particular process (includes I/O, CPU, memory time, waiting time in the ready queue)
- Maximize throughput – # of processes that complete their execution per time unit
- Maximize CPU utilization – the proportion of the CPU that is not idle
- Minimize response time – amount of time it takes from when a request was submitted until the first response is produced (interactivity)
- Waiting time: time spent in the ready queue
- Fairness: avoid starvation

First-Come, First-Served (FCFS) Scheduling

Process	CPU Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The schedule is:



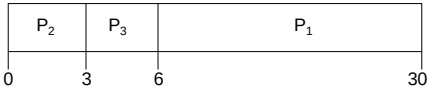
- Turnaround time for P_1 = 24; P_2 = 27; P_3 = 30
- Average turnaround time: (24 + 27 + 30)/3 = 27

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1 .

- The schedule is:



- Turnaround time for P_1 = 30; P_2 = 3; P_3 = 6
- Average turnaround time: (30 + 3 + 6)/3 = 13
- Much better than previous case.
- Short process delayed by long process: *Convoy effect*

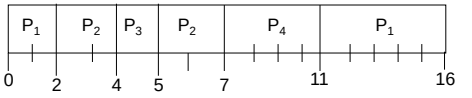
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its CPU time. Use these lengths to schedule the process with the shortest CPU time
- Two variations:
  - Non-preemptive – once CPU given to the process it cannot be taken away until it completes
  - preemptive – if a new process arrives with CPU time less than remaining time of current executing process, preempt
- Preemptive SJF is optimal – gives minimum average turnaround time for a given set of processes
- Problem:
  - don't know the process CPU time ahead of time

Example of Preemptive SJF

Process	Arrival Time	CPU Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)

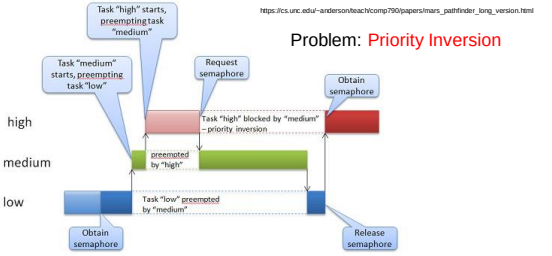


- Average turnaround time = (16 + 5 + 1 +6)/4 = 7

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted CPU time
- Problem: Starvation – low priority processes may never execute
- Solution: Aging – as time progresses, increase the priority of the process

What Happened on the Mars Pathfinder (1997)?



https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

Solution: Priority Inheritance [*L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.]*

Disclaimer

- Parts of the lecture slides were derived from those by Kai Shen, Willy Zwaenepoel, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).